UNIVERSITY OF THESSALY

DIPLOMA

Comparative performance analysis of Vulkan and CUDA programming model implementations for GPUs

---

Author:
Bodurri KLAJDI

Supervisor:
Christos D. ANTONOPOULOS
Examiners:
Nikolaos BELLAS
Spyros LALIS

A thesis submitted in fulfillment of the requirements for the degree of Diploma

in the

Computer Systems Laboratory (CSL)
Department of Electrical and Computer Engineering

Volos, June 2019

"The science of today is the technology of tomorrow."

Edward Teller

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Περίληψη

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

Συγκριτική ανάλυση της επίδοσης υλοποιήσεων των μοντέλων προγραμματισμού **Vulkan** και **CUDA** για **GPUs**.

Κλάιντι Μποντούρρι

Η defacto λύση πλέον για εκτέλεση πολλών παράλληλων και μεγάλων υπολογισμών είναι η χρήση GPU. Ο παραπάνω ισχυρισμός οφείλεται στον εγγενή σχεδιασμό των GPUs που αποτελείται από πάρα πολλούς πυρήνες που μπορούν να εκτέλεσουν πολλές πράξεις παράλληλα μεταξύ τους. Όμως, η χρήση τους για υπολογισμούς δεν ήταν εύκολη μέχρι πριν την δημιουργία κάποιων συγκεκριμένων API όπως CUDA, OpenCL καθώς και του πιο πρόσφατου Vulkan. Στην παρούσα διπλωματική εργασία εξετάζουμε το Vulkan API υλοποιώντας τον αλγόριθμο συνέλιξης εικόνας. Έχοντας τον ίδιο αλγόριθμο σε CUDA και συγκρίνοντας τις δύο υλοποιήσεις μπορούμε να δούμε τα πλεονεκτήματα αλλά και τους περιορισμούς του Vulkan μιας και πρόκειται για ένα πολύ καινούργιο API. Από αυτήν την σύγκριση επίσης φαίνεται η ανάγκη ανάπτυξης εργαλείων όπως compilers και profilers και αυτό συμβαίνει γιατί η απόδοση του αλγορίθμου εξαρτάται πάρα πολύ από τον compiler που χρησιμοποιείται και όχι από το API. Βλέποντας την τεράστια διαφορά στον χρόνο εκτέλεσης απο τις δύο υλοποιήσεις, εξετάζουμε γιατί η υλοποιήση σε CUDA είναι κατά πολύ γρηγορότερη και προσπαθούμε να χρησιμοποίησουμε κάποια υπάρχοντα profiling tools και compilers ώστε να βελτιστοποιηθεί ο χρόνος εκτέλεσης της Vulkan υλοποίησης. Τέλος, μέσα απο την υλοποίηση της συνέλιξης εικόνας δείχνουμε πόσο εύκολη ή δύσκολή είναι η υλοποίηση εφαρμογών σε Vulkan και διάφορα εργαλεία και πρακτικές που βοηθούν στην βελτιστοποίηση του χρόνου εκτέλεσης των εφαρμογών.

UNIVERSITY OF THESSALY

Abstract

Department of Electrical and Computer Engineering

Diploma

Comparative performance analysis of Vulkan and CUDA programming model implementations for GPUs

by Bodurri KLAJDI

The de facto solution for massively parallel calculations is the use of GPU. This stems from the inherent design of GPUs, which consists of a large number of cores capable of executing operations in parallel with each other. However, their use for computation was not easy until some specific APIs such as CUDA, OpenCL, and the recent Vulkan. In this thesis, we investigate the performance of Vulkan API by implementing the Image Convolution algorithm. By also implementing the same algorithm in CUDA and comparing the two implementations, we can identify the advantages and disadvantages of the recent Vulkan API. This comparison reveals the need to develop tools such as compilers and profilers as the performance of the algorithm is defined by the compiler and not the API. Observing the substantial discrepancy in the execution time of the two implementations, we explore why the CUDA implementation is noticeably faster. We then try to use some existing profiling tools and compilers in order to optimize the execution time of the Vulkan implementation. Finally, by implementing the Image Convolution, we highlight the difficulty of implementing compute applications in Vulkan. We also recommend tools and practices which optimize the execution time of the applications.

Acknowledgements

I would first like to thank my thesis advisor Prof. Christos D. Antonopoulos. The door to Prof. Antonopoulos office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work but steered me in the right direction whenever he thought I needed it. I would also like to acknowledge Prof. Nikolaos Bellas and Prof. Spyros Lalis as the readers of this thesis, and I am gratefully indebted for their very valuable comments on this thesis.

I would also like to thank the members of the CLS lab and especially Maria Gkeka who provided me with a friendly work environment, and they helped me conducting this thesis.

Finally, I must express my very profound gratitude to my parents and to my beloved friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Background

Since the early ’60s, when the computer industry first used the term "central process unit" (CPU), increasing the performance of processors at the hardware level was the main objective for engineers. Some of the techniques which have been successfully applied with the purpose of increasing the performance of processors were pipelined CPU in order to increase the instruction throughput, multilevel caches to hide memory latency, instruction level parallelism to measure how many of the instructions in a computer program can be executed simultaneously and superscalar processors for executing multiple instructions during a clock cycle.

With the methods above reaching their limitations, the need for increasing the CPU performance even further led the industry to the development of multicore processors. The introduction of multicore CPUs brought a new form of parallelism since some regions of an application could run concurrently on different cores of the same CPU. The impact of multicore CPUs in the speedup of parallelizable applications was considerable, but the limited number of cores mainly restricted the performance improvement.

On the other hand, Graphics Processing Units (GPUs) are designed with many cores in order to display computer graphics to the screen. The individual cores of a GPU are slower and simpler than the core of a CPU, but they provide high total throughput. Additionally, because of the highly parallel structure of GPUs, it makes them more efficient than CPUs for algorithms that process large blocks of data in parallel. As a result, the scientific community introduced the term "General-purpose computing on graphics processing units" (GPGPU). GPGPU is the use of a GPU to perform computations traditionally handled by the CPU. In the beginning, GPGPU was not widely well received until the release of CUDA API by NVidia [5] and OpenCL API from Apple/Khronos Group [15] which can leverage the speed of a GPU without requiring full and explicit conversion of the data to a graphical form.

Despite the release of the APIs as mentioned earlier for GPUs which brought the scientific community closer to GPGPU, there were many development tools yet to be developed in order for programmers to fully exploit the capabilities of GPUs for computing. Nvidia initially released CUDA almost 12 years ago. During that time, a toolkit has been developed which provides the capability of developing high-performance GPU-accelerated applications. The CUDA Toolkit includes features such as a very efficient compiler [3], GPU-accelerated libraries [17][6][1], debugging, and most importantly, profiling tools [4][20][16]. However, the CUDA Toolkit is proprietary, meaning we can take advantage of the tools above only on Nvidia specific hardware.

OpenCL launched almost ten years ago. While CUDA can run only on Nvidia specific hardware, OpenCL came to solve this fragmentation of programming models for different devices. OpenCL is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs, and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms. Despite the functional portability that OpenCL promises, it does not promise performance portability [7].

Although OpenCL and CUDA are mainly developed and used for GPU computing, we should not forget the purpose of GPUs, which is graphics rendering. In order for graphics developers to achieve hardware-accelerated rendering, Silicon Graphics released OpenGL 26 years ago [14]. OpenGL is a cross-language, cross-platform API for rendering 2D and 3D vector graphics. It has become the most widely-used open graphics standard in the world. The leading developer of OpenGL right now is Khronos Group.

In 2015, Khronos Group announced another API, the Vulkan API [10] along with SPIR-V [12]. Vulkan is a close-to-metal graphics and compute API that provides high-efficiency, low overhead, cross-platform access to modern GPUs used in a wide variety of supported devices from PCs and consoles to mobile phones and embedded platforms. Vulkan is supposed to provide numerous advantages over other GPU APIs such as CUDA, OpenCL, and OpenGL.

More specifically, Vulkan tries to combine the functionality of OpenGL and OpenCL into one API. It has been designed with multithreading in mind, which eliminates the single-core limitation that OpenGL is faced with. Its parallel characteristics allow asynchronous workload generation from multiple CPU threads that only start executing on the GPU after the submission. Its close-to-metal nature provides finer-grained control over GPU resources. As such, the developer is responsible for synchronization, memory allocation, and work submission that results in lower driver overhead compare with other approaches.

While in CUDA, several operations are handled by the driver, the performance gains

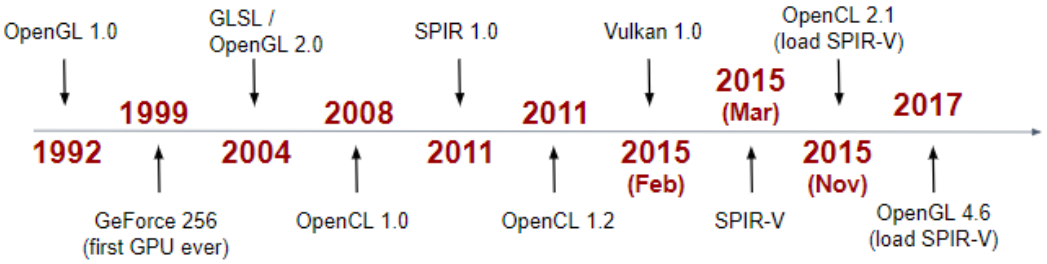


FIGURE 1.1: Khronos Group through time. [8]

of Vulkan due to lower driver overhead are offset by the additional development effort and time needed to implement an application. Also, Vulkan lacks development tools such as a potent compiler and profiler that CUDA has due to its maturity.

1.2 Problem statement and Contributions

Many video game companies are already using Vulkan for graphics. However, using it for computing is still challenging due to the verbosity of the API. In order to evaluate if Vulkan is viable for computing, the following questions need to be answered:

- What level of performance can Vulkan achieve?
- What is the relative performance of Vulkan compare with CUDA?
- How difficult is it to implement a compute application in Vulkan?
- What is the status and the level of maturity of the available Vulkan toolchain?

In the current thesis, we try to answer these questions in the following ways:

- We implement a simple application in order to show the potential of Vulkan considering its close-to-metal API. Since developing an application in Vulkan includes some standard steps such as creating a logical device, a pipeline, reading the kernel code and memory allocation, we provide a library providing the functionality as mentioned above.
- We implement the Image Convolution algorithm both in Vulkan and CUDA. We show what kind of performance the implementation of the image convolution on Vulkan can achieve and the performance loss in comparison with the implementation on CUDA. While the performance of the image convolution on CUDA is higher in many circumstances, we also observe Vulkan outperforming CUDA on equal terms by disabling the optimizations for CUDA that Vulkan lacks due to it being a recently released API.
- We show that the performance of image convolution has less to do with the API that is used to implement it and more with the lack of optimized development

tools that are being used for Vulkan. We analyze the primarily used compiler for both APIs and observe that glslang(Vulkan) [13] mostly provides graphics oriented optimizations. On the other hand, NVCC(CUDA) is highly optimized for a wide array of use cases, including compute, which is the main focus of this thesis.

- Given that a GPU executes a binary, we attempt to optimize the Image Convolution implementation on that level. Considering the limitations of glslang, we try to port the binary resulting from glslang to NVCC, which has proven optimizations capabilities.
- Finally, the latest iteration of CUDA(10.1) supports Vulkan interoperability. In this case, Vulkan is mainly used for graphics rendering. We, however, try to exploit the interoperability of CUDA with Vulkan for computing purposes.

1.3 Thesis structure

Chapter 2 shows the capabilities of Vulkan API through a vector addition example. Moreover, it shows the toolchain we use for the development and the compilation of Vulkan applications throughout the current thesis.

Chapter 3 describes the Image Convolution algorithm and the implementation of it in Vulkan and CUDA.

Chapter 4 presents the performance of Image Convolution in Vulkan and CUDA. Additionally, we explore the reason behind the significant performance of Image Convolution in CUDA through NVVP, which is a visual profiler for CUDA applications.

Chapter 5 investigates the possibilities of whether the binaries which are generated by the Image Convolution application in Vulkan contain machine code.

Chapter 6 continues the investigation from Chapter 5 on the binaries. In more detail, we try to convert the binaries from the Vulkan application to LLVM IR, which is a popular intermediate representation. We also implement the Image Convolution in OpenCL and then we convert it to SPIR-V, in order to improve the performance of the application in Vulkan.

Chapter 7 concludes with a summary of our main findings from the thesis and presents a direction for future work.

Chapter 2

A Simple Compute Application

2.1 Overview

In this chapter, we are going to show the main components of the Vulkan API by developing a simple compute application. The application is a vector addition and is very easy to implement. However, before we dig into the details of the implementation, we summarize the process of developing a Vulkan application on the following steps:

- Create an instance, pick a physical device, create a logical device, and pick a queue.
- Allocate memory and buffers for the application.
- Write the code for the compute shader on GLSL and compile it to SPIR-V.
- Create descriptor sets and compute pipelines.
- Create command buffer, record commands to it.
- Dispatch the command buffer.
- Evaluate the results.

In the next sections, we are going to discuss and present the Vulkan API calls for each of the above steps. However, the current chapter cannot be used as a replacement of the Vulkan API specs. For example, we are not mentioning which are the valid arguments for Vulkan functions. We want to show the process flow of implementing an application on Vulkan.

2.2 Instance, Devices, and Queues

As we see in Figure 2.1, according to hierarchy of Vulkan, the the first thing we should do is to create an *instance*. The *instance* is an object that initializes the Vulkan

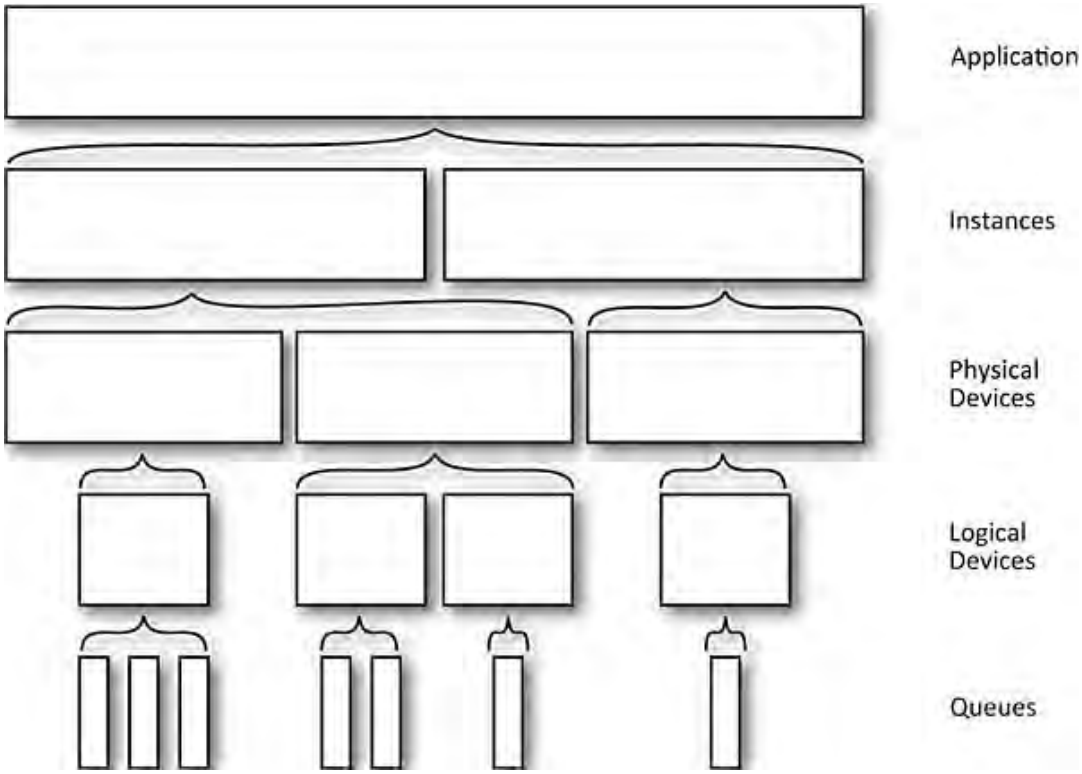


FIGURE 2.1: Vulkan Hierarchy of Instance, Device, and Queue. [21]

library and keeps the state of the application. To create an instance object, we call the *vkCreateInstance()* function.

Once we are done with it, we are about to query all the available *physical devices* on our system and pick one or more of them. A physical device is the hardware (e.g., CPU or GPU) that the application is going to run on. So, we have to discover all the physical devices on the system. We can do that by calling the *vkEnumeratePhysicalDevices()* function twice. The first time we call that function is to find out the number of available physical devices on our system. Then, we allocate an array of *VkPhysicalDevice* with the size of that number. Finally, we call the function *vkEnumeratePhysicalDevices()* for the second time, which now fills the array we allocated with pointers of physical devices. At this point, we are done querying the physical devices of our system.

Later, if our system has many physical devices, we have to pick only the devices which are suitable for our application. Each physical device has some properties to offer. In this example, we are interested only in the *queue family* properties. A physical device has many *queues* which are categorized based on their capabilities in groups called *queue families*. A *queue* is where the workload is submitted. Each *queue family* supports one or more capabilities such as *graphics*,*compute*, *transfer* operations and *sparse binding* memory management operations. Since we are building a compute application, we are interested only in queue families that can support compute operations. In order to pick the proper physical device that supports queues with

computing properties, we do the following. We iterate through the array of physical devices we retrieved before, and for each physical device we query for *queue family properties* by calling the *vkGetPhysicalDeviceQueueFamilyProperties()* function. This function retrieves all the queue family properties for a physical device and stores them into an array. Then, we check if there is a queue family in the array which supports compute operations and we save its index because we are going to use it later.

After completing the process of discovering and picking a proper physical device for our application, we should create a *logical device*. A logical device represents the connections to the physical device (e.g., resources) and the application interacts mostly with it. We can create more than one logical device for a physical device, but we are not going to need it here. The process to create a logical device is done by calling the *vkCreateDevice()* function.

Lastly, we have to allocate a *queue*. We have already mentioned what a queue is and why it is essential to select a queue with properties that fit the application's needs. On the paragraph above, we found the index of a family queue with computing capabilities, so now it is time to allocate a queue from that family. We do that by calling the *vkGetDeviceQueue()* function and giving as argument the index of the family queue we are interested in.

In summary, we created an instance that initializes the Vulkan library, and we picked a physical device to run our application and created a logical device which connects the physical device with the application. Finally, we allocated a queue that we are going to submit the workload of the application.

2.3 Compute Shader on GLSL

In this section, we are going to discuss what is a compute shader and how to write a shader using GLSL.

A compute shader is the parallel code which is going to run on the GPU. In our case, the compute shader is the code for the vector addition. As we can see at Listing 2.1, the vector addition is straightforward, and it is written in GLSL.

GLSL is a shading language for OpenGL. Looking at the code, we can say that GLSL is not tricky to use for compute shaders because of its C language syntax. One thing we should make sure to remember from Listing 2.1 is the binding numbers we gave to the vectors because it is something we are going to need later.

However, we cannot feed the Vulkan API with raw GLSL code as it requires SPIR-V binary. To achieve this, we use the glslangValidator [13] in order to compile the GLSL compute shader as SPIR-V binary. After the compilation, we load the SPIR-V binary into the application, we create a *vkShaderModule*, and we initialize it with

the shader binary. Later, we will bind this shader module to a *compute pipeline*. The Figure 2.2 shows the above process.

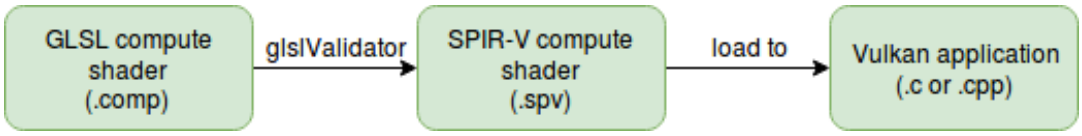


FIGURE 2.2: Generation of SPIR-V shader and binding with Vulkan application.

```
#version 430

layout (std430, binding=0) buffer srcBuf1 {
    double data[];
} vector1;

layout (std430, binding=1) buffer srcBuf2 {
    double data[];
} vector2;

layout (std430, binding=2) buffer destBuf {
    double data[];
} resultVector;

layout(local_size_x = 32, local_size_y = 1, local_size_z = 1) in;

void main()
{
        int x;
        x = int(gl_GlobalInvocationID.x);

        resultVector.data[x] = vector1.data[x] + vector2.data[x];
}
```

LISTING 2.1: Vector Addition with GLSL

2.4 Allocate Memory and Buffers

We had a brief look at how the vector addition is implemented with GLSL. From here, the next step is to allocate the memory resources for the implementation, but first, we have to explain the *memory allocation* and *buffers* on Vulkan.

On Vulkan, we define the terms *host memory* and *device memory*. The main difference between these two is that the host memory is accessed and used from the CPU while the device memory is accessed from the physical device that we will run the compute shader. The device memory can have some properties and types. For example, one type of memory can be *HOST VISIBLE*, which specifies that memory can be

mapped for host access. Other types are *DEVICE LOCAL* or *HOST COHERENT* and several others. The application is responsible for allocating memory with the proper type based on its need. Vulkan gives this flexibility for better memory performance and utilization. In our application, we need three vectors, so three memory allocations with preferred memory type of both *HOST VISIBLE* and *HOST COHERENT*. We need these two types so we can evaluate the results of the vector addition on the host side after the workload is done. We can get the memory properties of a physical device by calling the *vkGetPhysicalDeviceMemoryProperties()* function. We iterate through all different memory types, and we try to find a memory which supports host visibility and coherency at the same time. After finding a suitable memory, we can allocate it by calling *vkAllocateMemory()*.

However, Vulkan cannot operate on the memory we just allocated. It operates on resources which are backed by memory. Vulkan supports two types of resources. The first one is *buffers*, which are unstructured arrays of bytes. The other type of resources is *images*. The images contain format information, and they are structured. In this example, we will use only buffers.

First, we have to create three buffers, because we did three memory allocations, one for each vector. We create the buffers by calling the *vkCreateBuffer()* function. We also need to bind the buffers to the allocated memory. We can do that with *vkBindBufferMemory()*.

2.5 Compute Pipeline and Descriptor Sets

The pipeline concept is mainly used in computer graphics. A *graphics pipeline* describes what are the steps for a graphics system in order to render a 3D or 2D object on the screen [23]. Apart from graphics pipeline, Vulkan supports *compute* and *ray tracing* pipelines. In this example, we are only interested in compute pipelines.

Before we create a compute pipeline, we should define the *pipeline layout* and bind the allocated resources through the *descriptor sets*. A descriptor set is a set of resources that is used to bind the resources to the pipeline. For each descriptor set, we should define its layout, which describes the types of resources in the set. First, we create a descriptor pool which is responsible for allocating descriptor sets. A descriptor pool is created through *vkCreateDescriptorPool()*. Then, we define the descriptor set layout by using the *VkDescriptorSetLayoutBinding* structure and we can allocate it through *vkCreateDescriptorSetLayout()*. After that, we allocate a descriptor set by calling the *vkAllocateDescriptorSets()* function.

Once we complete the descriptor set allocation, we are ready to bind the allocated resources to the descriptor set. We can do that by using the *VkDescriptorBufferInfo*

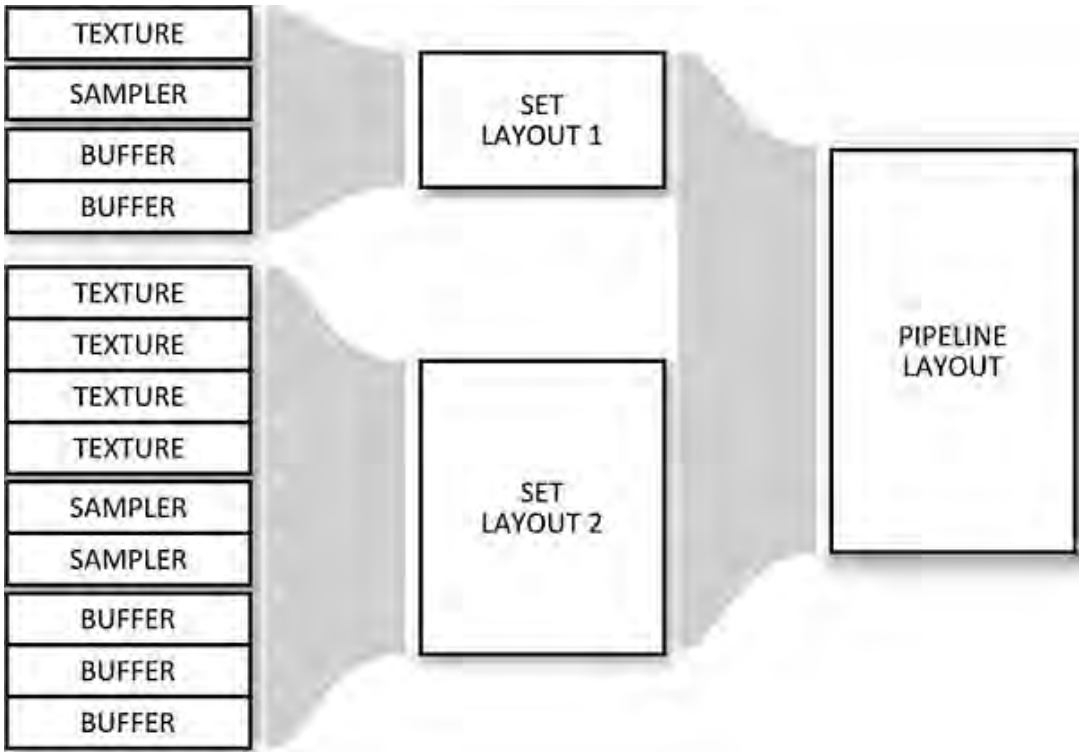


FIGURE 2.3: Relationship between resources, descriptor set and pipeline. [21]

and *VkWriteDescriptorSet* structures. In order for the binding of resources to be correct, we should use the same binding numbers as the numbers from the GLSL code we used before. After this, we call the *vkUpdateDescriptorSets()* function.

Although the resources are now bound to the descriptor set, the descriptor set alone cannot be used in the pipeline. A *pipeline layout* is needed in order to enable the pipeline to access the descriptor sets and by extension, the resources. The relationship between the descriptor set layout, and pipeline layout is shown in Figure 2.3.

We define the pipeline layout through the *VkPipelineLayoutCreateInfo* structure. In our example, we have three buffers which are bound in one descriptor set so the pipeline layout will bind to one descriptor set. After the definition of the layout, we allocate it by calling the *vkCreatePipelineLayout()* function.

After completing the aforementioned tasks, the descriptor set is connected with the resources, and the pipeline layout is connected with the descriptor set. Now, we can create a compute pipeline. We can do that by calling the *vkCreateComputePipelines()*.

2.6 Command Buffers

The construction of the compute pipeline is completed, but it cannot run on the physical device yet. We have to bind it to a *command buffer*. A command buffer can

record commands which will later be submitted to a queue for execution. Some of the commands can be *pipeline-binding to the queue* or *resource-binding to the queue*.

Before we allocate a command buffer and record commands to it, we should create a *command pool* which will be used to allocate a command buffer. We do that by calling the *vkCreateCommandPool()* function. Then, we allocate a command buffer through *vkAllocateCommandBuffers()*. Now we are ready to record commands. We start the recording by calling the *vkBeginCommandBuffer()*. In our example, we will record three commands. The first one is the binding of the pipeline to the queue by calling the *vkCmdBindPipeline()* function. The second one is to bind the resources to the queue. We can do that by calling the *vkCmdBindDescriptorSets()* function. The last command is the compute shader invocation on the GPU. We can do that by calling the *vkCmdDispatch()* function. This function declares the geometry of the shader. In other words, it declares the number of threads that will execute the compute shader. In the end, we can stop recording commands to the command buffer by calling the *vkEndCommandBuffer()* function.

2.7 Workload Submission and Result Evalution

In order for the vector addition to run on the GPU, we should submit the command buffer to the queue. In section 2.2, we allocated a queue from the queue family. Subsequently, we can call the *vkQueueSubmit()* function which will submit the command buffer to the queue, and it will start executing the pre-recorded workload immediately. This function is non-blocking, which means that it doesn't wait for the submitted workload to finish. To solve this problem, we call the *vkQueueWaitIdle()* function. The above function waits for the queue to complete its assigned job.

Once the *vkQueueWaitIdle()* returns, the vector addition is completed. But, the results cannot be accessed yet. The results are still in the device memory. We have to map the device memory to the host memory. We do that by calling the *vkMapMemory()*. Then we can evaluate the results. After evaluation, we should unmap the memory with *vkUnmapMemory()*.

At the end of the process, we should not forget to free every resource we used. For example, we have to free the command buffer, the command pool, the pipeline, and the buffers. There are functions we can use like *vkFreeCommandBuffers()*, *vkDestroyPipeline()* and *vkDestroyBuffer()*.

2.8 Summary

We created an instance on Vulkan, and we picked a physical device from our system. Furthermore, we showed how to implement a simple vector addition on GLSL, and

we compiled it to SPIR-V binary. Then, we loaded the binary to the application, we allocated the appropriate buffers on Vulkan, and we backed them with the allocated memory. Additionally, we created a compute pipeline, and we recorded a command buffer. Lastly, we submitted the command buffer to a queue, in order for the vector addition to run on the physical device we picked.

Vulkan is verbose API, and that has been made apparent by this example. However, because of its verbosity, it can fully utilize and exploit the characteristics of the system it runs on. We demonstrated that in this example, by having to pick the right type of queue and the right type of memory. Vulkan offers low driver overhead, but everything comes with a price.

In conclusion, Figure 2.4 summarizes the process of developing a compute application on Vulkan. This process is almost the same for every compute application on Vulkan except for some more complex applications. In this thesis, based on the above steps, we have implemented a simple library which can handle some of these Vulkan operations. For example, the library supports instance initialization, physical device picking, and memory allocation. Additionally, it supports buffer binding, shader loading, and binding, and finally, descriptor set creating and binding. All of the above functionalities are needed in order to implement a Vulkan compute application.

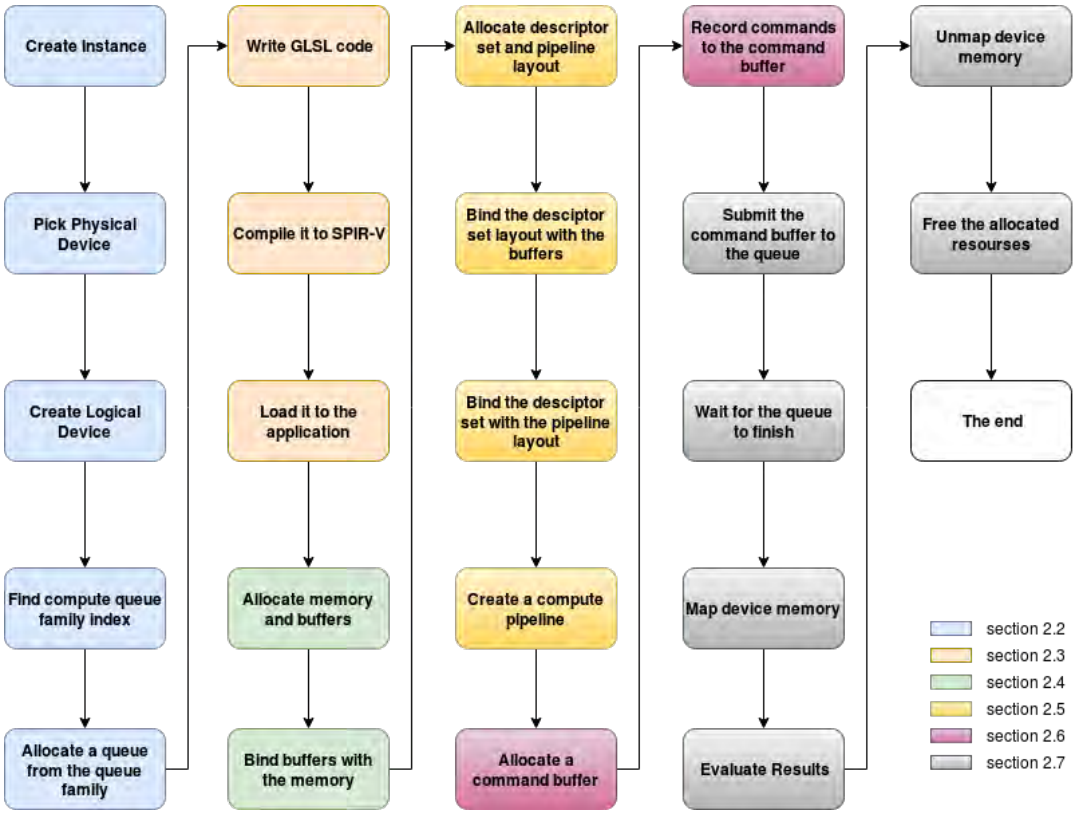


FIGURE 2.4: Steps for building a Vulkan Compute Application.

Chapter 3

Image Convolution

3.1 Image

In computer terms, digital images are represented as an array of numbers (*pixel grid*). Each of these numbers describe the *intensity* of the *pixels*. A pixel is the smallest addressable element in a display. The intensity of a pixel can describe the color or the brightness of the pixel. Figure 3.1 shows an example of how an image can be represented in a computer system. Furthermore, it shows the intensity values of some pixels in a digital image.

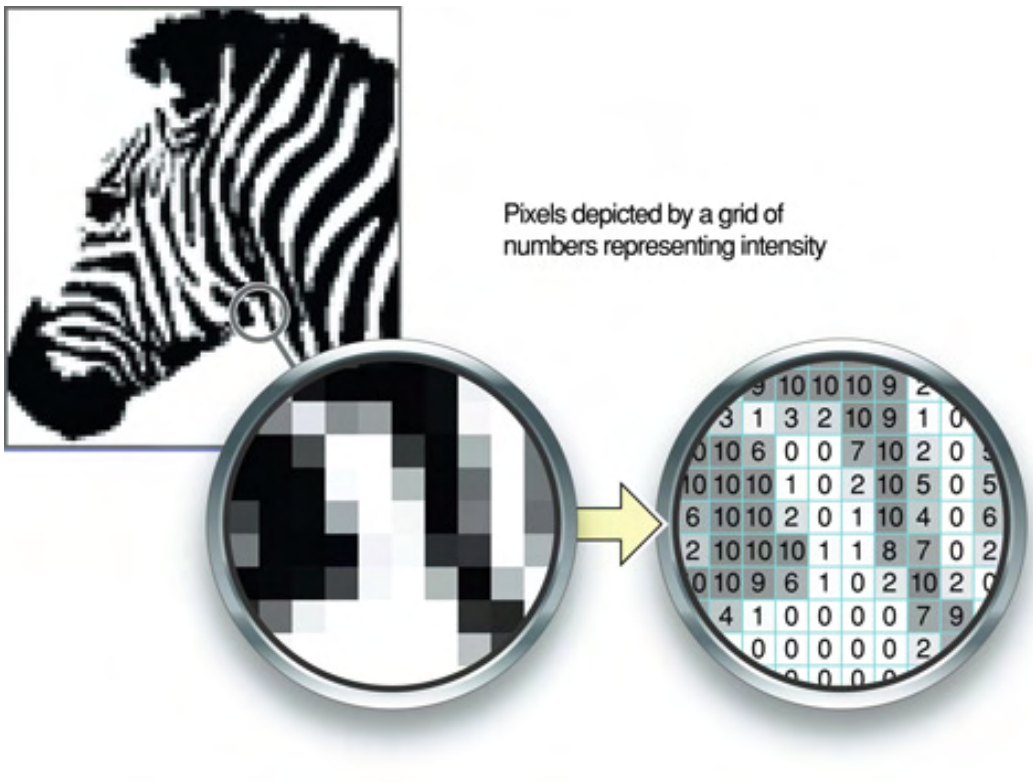


FIGURE 3.1: Digital representation of an image. [11]

Since the intensity of a pixel in an image is represented as a single number, it allows to process this intensity value and therefore to process the whole image. Considering

this, computer science introduced the term *Digital Image Processing*. Digital image processing is a method to perform operations on an image in order to extract useful information from it [24].

There are many operations in image processing. For example, *edge detection*, *blur*, *sharpen* and several others. Figure 3.2 and 3.3 show some of the operations we mentioned above. Specifically, in Figure 3.3 we see the term *kernel*. Kernel is a matrix which can be *multiplied* with an image and results to a transformed new image. This *multiplication* is called *Image Convolution* and is the algorithm we are going to develop in Vulkan and CUDA in the current thesis.

(A) Before processing.

(B) After image process operation.

FIGURE 3.2: Negative transformation of an image.

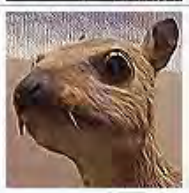
Operation	Kernel ω	Image result g(x,y)
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	
Gaussian blur 5 × 5 (approximation)	$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	
Unsharp masking 5 × 5 Based on Gaussian blur with amount as 1 and threshold as 0 (with no image mask)	$\frac{-1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & -476 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$	

FIGURE 3.3: Basic Image Processing operations. [25]

3.2 Algorithm Explanation

The core algorithm of the aforementioned image processing operations is the image convolution. Only the applied kernel is different in these operations. The kernel, based on its values, can give various effects on images. However, the method (Image convolution) to get the transformed image is the same. As we already mentioned, we *multiply* the image with the kernel. This multiplication is not the same neither as simple as the basic math multiplication we know. Figure 3.4 shows how Image Convolution is applied for a pixel.

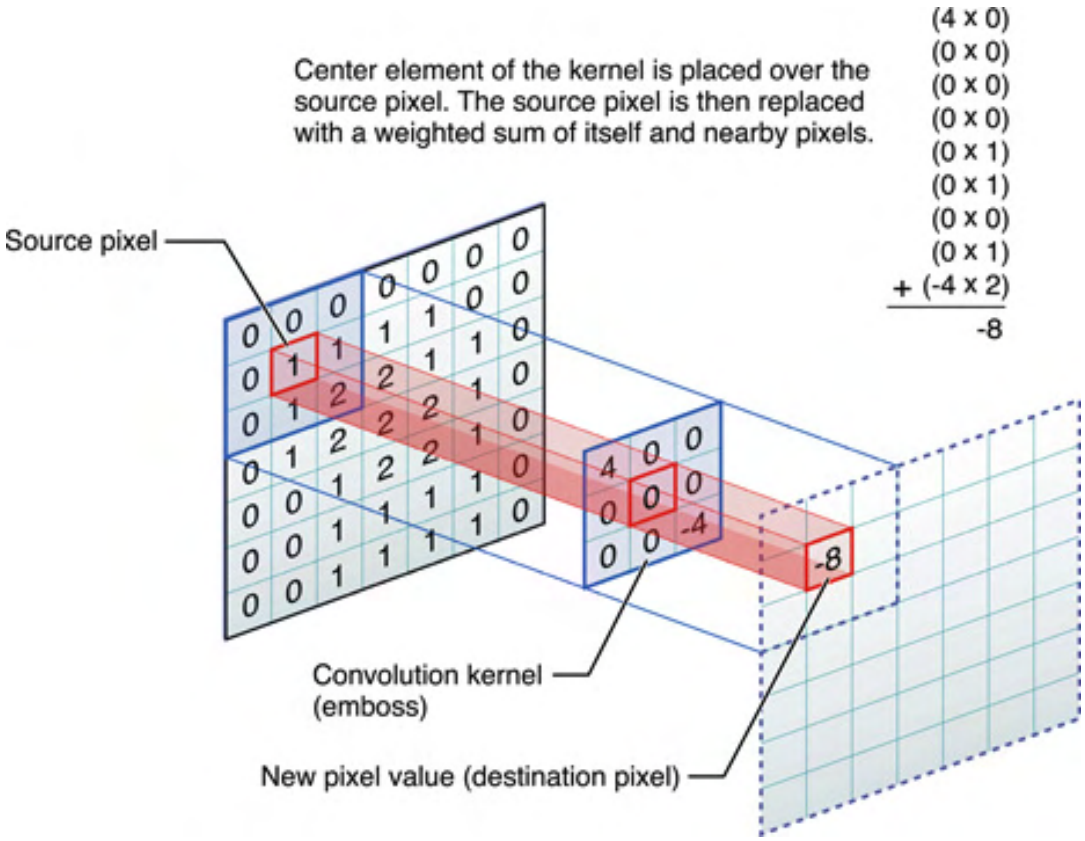


FIGURE 3.4: Image Convolution for a pixel. [11]

For each pixel, we add the weighted intensity values of its neighbor pixels. This weighted sum is the new intensity value of the pixel. According to this, we can understand that the convolution process for each pixel can be done separately from the convolution of the other pixels. If our computer system supports it, we can calculate the convoluted intensity value for each pixel of the whole image concurrently. For example, if we have a digital image with 1920x1080 pixels, and we want to apply the image convolution algorithm to it, we can assign the 1920x1080 pixels to 1920x1080 compute threads. Each of these threads will calculate only the intensity value of their assigned pixel. As a result, the time to calculate the convoluted intensity value of all pixels of the image is roughly the same as the time to calculate the convoluted intensity value of one pixel since all the threads can run in parallel.

The drawback of the above example is the number of threads that is required in order to run the Image Convolution fully parallel. For big images, the amount of threads is quite huge. No typical CPU can support such numbers of threads. On the other hand, GPUs can handle this very efficiently since they are designed for this purpose. This is the reason we chose to implement Image Convolution on GPU by using Vulkan and CUDA. Image Convolution can run on GPUs, and because of this, we can get a good view of how Vulkan and CUDA perform.

3.3 Implementation on Vulkan and CUDA

3.3.1 Separable kernel

Before we dig into the technical details of the Image Convolution implementation on Vulkan and CUDA, we should mention that we are not trying to optimize the Image convolution algorithm. Instead, we are testing the relative performance of Vulkan compared to CUDA given an identical implementation.

Additionally, our Image Convolution implementation is using *separable kernels*. A separable kernel is a kernel which can be broken down into two 1-D kernels.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

(A) Separable kernel. (B) First 1-D kernel. (C) Second 1-D kernel.

FIGURE 3.5: A separable 2-D kernel and its corresponding 1-D.

In Figure 3.5, we can see a separable kernel and the equivalent 1-D vectors that we can extract from it. Following that, we can apply the Image Convolution algorithm on an image by using the two 1-D vectors, instead of the 2-D kernel. We apply the image convolution first on the rows of the image using the 3.5b kernel to produce an intermediate product. Then, we do the same on the columns of the intermediate image as shown in Figure 3.6 with the help of 3.5c kernel. The result will be the same as using the 2-D kernel.

Based on the above, our implementation is separated into two compute shaders(Vulkan) or two *kernels*(CUDA). The first compute shader is responsible for applying the image convolution over the rows of the image, and the second compute shader is responsible over the columns of the intermediate image. This holds true for the two CUDA kernels as well.

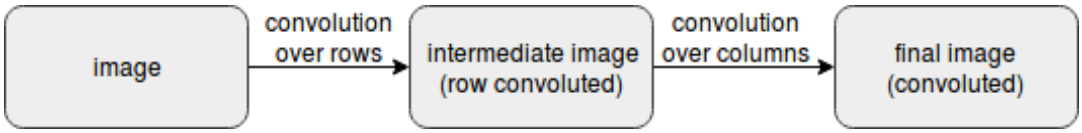


FIGURE 3.6: Image Convolution process.

We might get confused by CUDA kernels and by the term kernel from Image Convolution. To avoid confusion, we will refer to the kernel matrix from image convolution as *filter*, because it filters the digital image and gives an effect to it. On the other hand, CUDA kernel is a piece of code that is going to run on the GPU like the compute shader for Vulkan.

3.3.2 Image Convolution on Vulkan

As we mentioned in Chapter 2, we write the compute shaders in GLSL, and then we compile them in SPIR-V in order to be compatible with Vulkan. Listings 3.1 and 3.2 show the two compute shaders written in GLSL which we need for Image Convolution. We can see that they are very similar. Specifically, Listing 3.1 shows the Image Convolution over rows. The main functionality of the Image Convolution over rows is achieved inside the *for loop*. This *for loop* calculates the weighted sum for each pixel of the image by iterating the filter row-wise. The code from Listing 3.2 achieves the same functionality with a small difference. It iterates the filter column-wise. The combination of the two compute shaders implements the Image Convolution algorithm over an image.

Up to this point, we haven't mentioned Vulkan yet. We have explained the process we need to follow in order to implement an application on Vulkan in Chapter 2. The same process can be applied here as well with a slight difference. We split the Image Convolution implementation into two compute shaders. However, in Vulkan, we can bind a compute pipeline with only one compute shader. Consequently, we need two compute pipelines.

Furthermore, the sequence in which we record the commands in the command buffer must be defined carefully. In order to achieve the correct functionality of Image Convolution as shown in Figure 3.6, we must first record the compute pipeline which is responsible for the row-wise Image Convolution and then the other compute pipeline. Finally, we submit the command to the queue as we know and wait for the results.

```
#version 430

layout (std430, binding=0) buffer srcBuf {
    double data[];
} d_Input;

layout (std430, binding=1) buffer dstBuf {
    double data[];
} d_Output;

layout (std430, binding=2) buffer dstFilter {
    double data[];
} d_Filter;

layout (push_constant) uniform PushConstants {
    int imageW;
    int imageH;
    int filterR;
} convolutionInfo;

layout(local_size_x = 32, local_size_y = 32, local_size_z = 1) in;

void main() {
    int x, y;
    int k;
    int d;
    double sum = 0.0;

    x = int(gl_GlobalInvocationID.x);
    y = int(gl_GlobalInvocationID.y);

    for (k = -convolutionInfo.filterR; k <= convolutionInfo.filterR; k++)
    {
        d = x + k;
        if (d >= 0 && d < convolutionInfo.imageW) {
            sum += d_Input.data[y * convolutionInfo.imageW + d] *
                   d_Filter.data[convolutionInfo.filterR - k];
        }
    }
    d_Output.data[y * convolutionInfo.imageW + x] = sum;
}
```

LISTING 3.1: Convolution over rows in GLSL

```
#version 430

layout (std430, binding=0) buffer srcBuf {
    double data[];
} d_Input;

layout (std430, binding=1) buffer dstBuf {
    double data[];
} d_Output;

layout (std430, binding=2) buffer dstFilter {
    double data[];
} d_Filter;

layout (push_constant) uniform PushConstants {
    int imageW;
    int imageH;
    int filterR;
} convolutionInfo;

layout(local_size_x = 32, local_size_y = 32, local_size_z = 1) in;

void main() {
    int x, y;
    int k;
    int d;
    double sum = 0.0;

    x = int(gl_GlobalInvocationID.x);
    y = int(gl_GlobalInvocationID.y);

    for (k = - convolutionInfo.filterR; k <= convolutionInfo.filterR; k++)
     {
        d = y + k;
        if (d >= 0 && d < convolutionInfo.imageW) {
            sum += d_Input.data[d * convolutionInfo.imageW + x] *
                       d_Filter.data[convolutionInfo.filterR - k];
        }
    }
    d_Output.data[y * convolutionInfo.imageW + x] = sum;
}
```

LISTING 3.2: Convolution over columns in GLSL

3.3.3 Image Convolution on CUDA

We saw how Image Convolution is implemented on Vulkan. In this subsection, we describe how we can implement the same algorithm with CUDA. Listings 3.3 and 3.4 show the row-wise and column-wise Image Convolution CUDA kernel respectively. By comparing these two CUDA kernels with the two compute shaders above, we can see that they are similar. Especially the code inside the *for loop*, which implements the core functionality of the Image convolution, is practically the same for all the kernels and compute shaders. As a result, the factors that can affect the performance of Image Convolution on the GPU are the GPGPU API and the development tools we use. This should give us a view of how Vulkan performs in comparison to CUDA.

However, CUDA kernels are not ready to run on our system yet. First, we need to allocate memory for the two kernels and transfer the memory from the host side to the device side. Then we can run the two kernels. Finally, we return the results to the host side for evaluation. We followed the same process during the Image Convolution implementation on Vulkan.

The process of running the Image Convolution might be the same between CUDA and Vulkan, but they are different APIs, which means that we use different functions. In CUDA, in order to allocate memory we call the *cudaMalloc()* function. We need four memory allocations for our implementation. The first memory allocation is for the filter. We need also one memory allocation for the input image, one for the intermediate image and one for the output image. Once we are done with the memory allocation, we should fill this allocated memory with useful data. In other words, we should transfer the filter and the input image to the device side. We can do that by calling the *cudaMemcpy()* function. Now, we are ready to run the two kernels. In order for the Image Convolution to be correct, we first run the row-wise kernel, and then we wait for it to finish by calling the *cudaDeviceSynchronize()* function. Then, we run the column-wise kernel using as its input the output (intermediate image) of the row-wise kernel. We call the *cudaDeviceSynchronize()* function again in order for the column-wise kernel to finish as well. After this, we are done with Image Convolution on GPU. We should copy the result back to the host side by calling *cudaMemcpy()* function but with slightly different arguments. In the end, we evaluate the results and de-allocate the device memory by calling the *cudaFree()* function. Figure 3.8 summarizes the above steps.

The implementation is ready, but we have not compile the kernel code yet. In the current thesis, we use the NVVC compiler, which is developed by Nvidia. Figure 3.7 shows the compilation process.

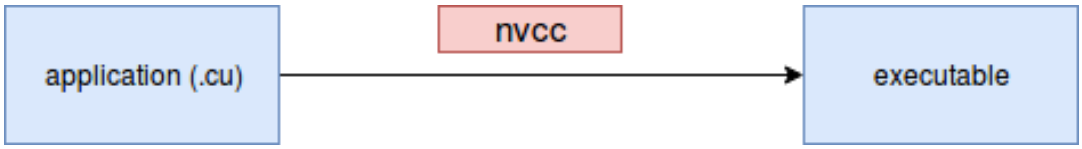


FIGURE 3.7: Compilation process of CUDA implementation.

```
__global__ void convolutionRowGPU(user_data_t *d_Dst, user_data_t *d_Src,
    user_data_t *d_Filter, int imageW, int imageH, int filterR) {
    int x,y,k;
    user_data_t sum=0.0;

    x = threadIdx.x + blockDim.x * blockIdx.x;
    y = threadIdx.y + blockDim.y * blockIdx.y;

    for (k = -filterR; k <= filterR; k++) {
      int d = x + k;
      if (d >=0 && d < imageW){
        sum += d_Src[y * imageW + d] * d_Filter[filterR - k];
      }
    }
    d_Dst[y * imageW + x] = sum;
}
```

LISTING 3.3: Convolution over rows CUDA

```
__global__ void convolutionColumnGPU(user_data_t *d_Dst, user_data_t *
    d_Src, user_data_t *d_Filter, int imageW, int imageH, int filterR) {
    int x, y, k;
    user_data_t sum = 0.0;

    x = threadIdx.x + blockDim.x * blockIdx.x;
    y = threadIdx.y + blockDim.y * blockIdx.y;

    for (k = -filterR; k <= filterR; k++) {
      int d = y + k;
      if (d >= 0 && d < imageW){
        sum += d_Src[d * imageW + x] * d_Filter[filterR - k];
      }
    }
    d_Dst[y * imageW + x] = sum;
}
```

LISTING 3.4: Convolution over columns CUDA

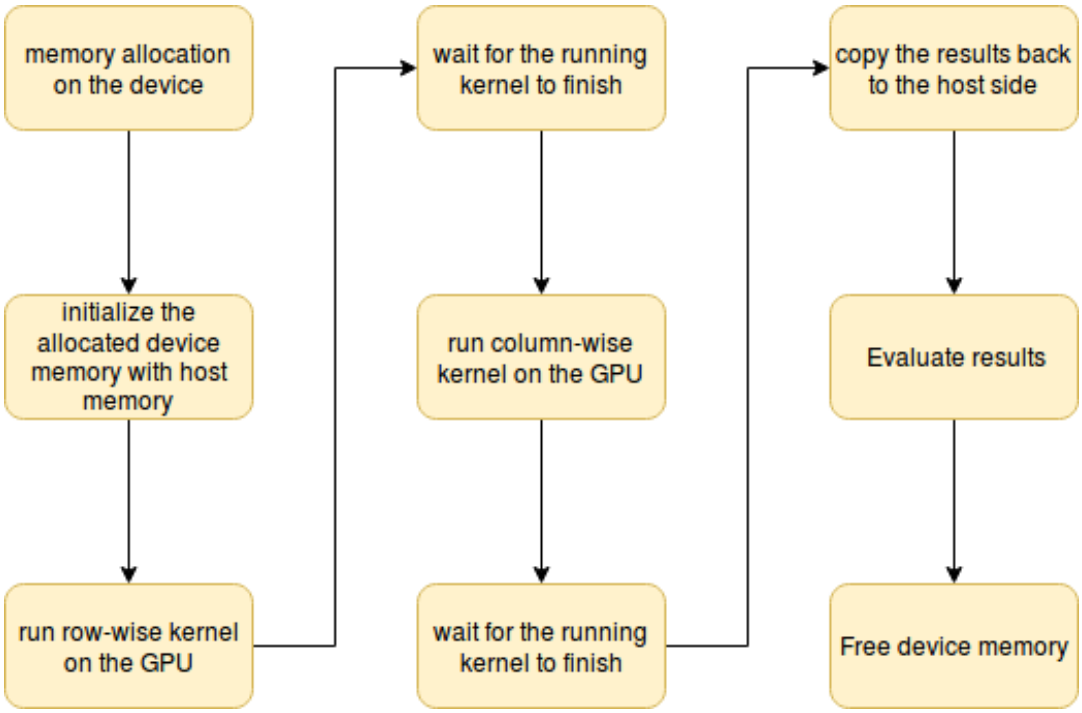


FIGURE 3.8: Steps for building the Image Convolution with CUDA.

3.4 Summary

In this section, we saw how digital images are represented on computers. We saw the importance of Image Convolution and why we chose to implement this algorithm in this thesis. Finally, we showed how to implement the same version of Image Convolution both in CUDA and Vulkan. In the next section, we will present the development tools we used in order to compile the Image Convolution in CUDA and Vulkan. Additionally, we will see the performance of Image Convolution on Vulkan and CUDA, and we will try to compare the performance of these two versions.

Chapter 4

Vulkan and CUDA Performance

4.1 GPU Characteristics

We have implemented the Image Convolution algorithm and we have described the toolchain we use for Vulkan and CUDA. As a next step, we execute the implementations and evaluate their performance on the GPU. Table 4.1 shows the characteristics of the available GPU in our system. The clock speed of the device is not essential because we want to show the relative performance of CUDA and Vulkan on the same device and not the performance of the device. One thing we should pay attention to, from Table 4.1 is the global memory size. The global memory creates a limitation on the size of images to which we want to apply the Image Convolution. We cannot apply the Image Convolution on images which exceed this size limitation.

GPU Driver	Nvidia 418.56
GPU Name	Nvidia GeForce GT 740M
Frequency	810-980 MHz
Global Memory	2 GB
Max dimension size per block	(1024, 1024, 64)

TABLE 4.1: GPU characteristics

Figure 3.7 shows that we can write the kernel code and the application in the same file (.cu), and then we can compile it with NVCC. On the other hand, the Vulkan compilation process is different. First, we write the compute shader code. Then we compile the compute shader with glslangValidator, and we load it to the application. Last, we use *gcc* in order to get the application executable.

We discussed the Image Convolution implementation in Chapter 3. We also showed the toolchain we used in order to compile the application for both Vulkan and CUDA. At this point, we are ready to run the algorithm and evaluate the relative performance.

4.2 Performance Evaluation

In this chapter, we will evaluate the total performance of the application for Vulkan and CUDA. The overall performance is the execution time of an application on the GPU. For example, some steps we have mentioned in Chapter 2 and 3 are the device initialization, device selection, memory copy to device side, and the computation time of the algorithm. All these steps are included in the total time.

Then, we will focus on the compute time of the Image Convolution in Vulkan and CUDA. We will show which implementation of Image Convolution (Vulkan or CUDA) performs better and the reason behind it.

4.2.1 Overall Performance

Figure 4.1 depicts, the total time of Image Convolution in Vulkan and CUDA. For small images such as 32x32 till 512x512, we can observe that the total time of the CUDA version is almost zero. On the other hand, in the Vulkan version, we always pay an overhead for creating an instance, for discovering, and picking a physical device. We do not pay this overhead in CUDA because CUDA is targeted only for Nvidia GPUs, and it is reasonable that the Nvidia drivers are highly optimized for it. In contrast, Vulkan is cross-platform, and as a result it is designed with in mind to run on every device. So, we need to query and pick a physical device for any system, and therefore, we have a non negligible overhead on the application. Figure 4.2 shows the percentage of the total time which is spent on computation and the percentage of total time, which is the overhead in order to run the application on the device successfully.

As we increase the image size, we can observe that the computation time begins to dominate the results. We will focus on the computation time in the next section, but for now, we can observe that the CUDA version can roughly achieve a *speedup of 2x* in comparison with the Vulkan version. The difference in total time between CUDA and Vulkan is because of the compiler that is used as we will discuss later.

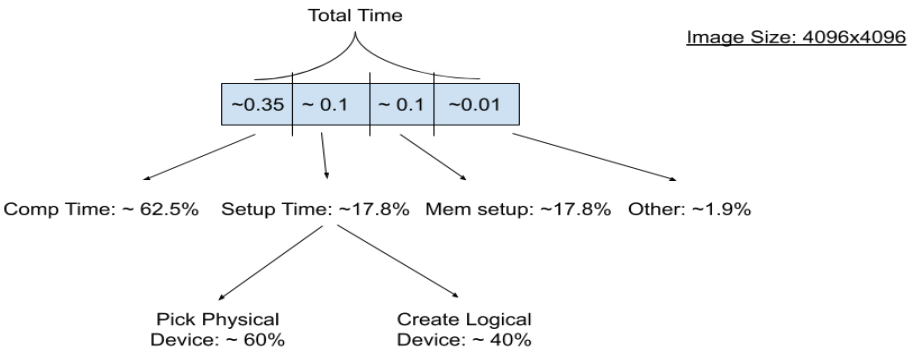


FIGURE 4.2: Execution time breakdown.

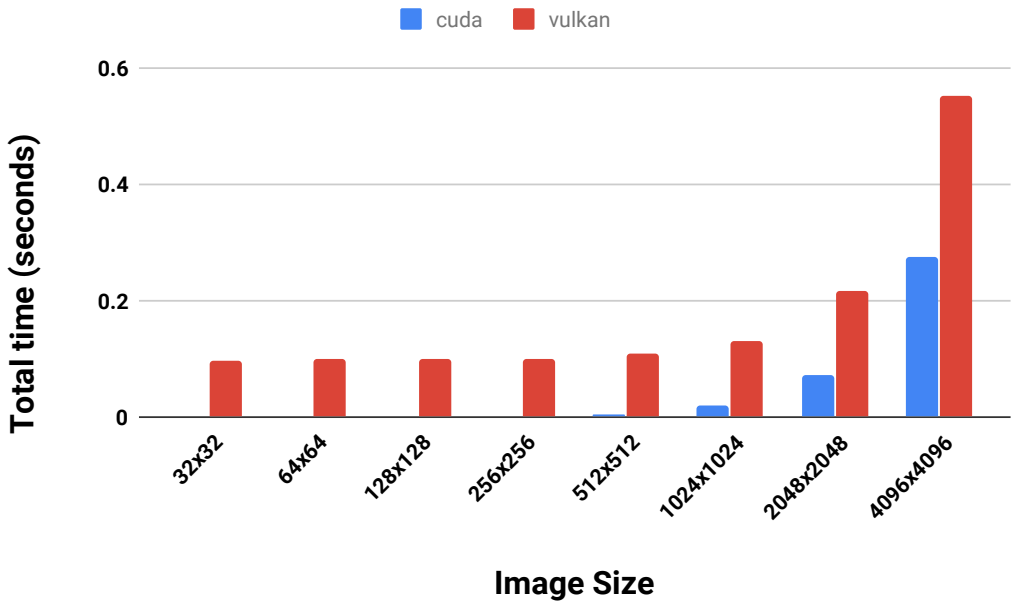


FIGURE 4.1: Total performance of Image Convolution using Vulkan and CUDA.

4.2.2 Computation Time

Before we analyze the computation time of Image Convolution in Vulkan and CUDA, we should mention what kind of optimizations a compiler can apply to our code. For Image Convolution in CUDA, we use the *NVCC* as a compiler. NVCC is a highly optimized compiler which supports four optimization levels (O0, O1, O2, O3). Each flag enables various optimizations. Some of the optimizations which can be applied are *loop unrolling for eliminating instructions that control the loop*, *instructions grouping of loads and stores to the global memory for higher memory bandwidth*, *assigning of registers to the most used variables of the code to reduce memory latency* and many more. The higher the optimization level number, the more advanced techniques the compiler will try to apply to our code.

While the NVCC is capable of applying various advanced optimizations in our Image Convolution code, the *glslangValidator* is not ready to support such optimizations for GLSL code. glslangValidator is responsible only for producing the SPIR-V binary from GLSL. We use another tool in order to apply optimizations to the SPIR-V binary (Figure 4.3). The tool is called *spirv-opt* and is provided by LunarG [9]. However, the spirv-opt is still under development, so it does not support as many advanced optimizations as NVCC for compute shaders. It is mainly used to reduce the SPIR-V binary file size.

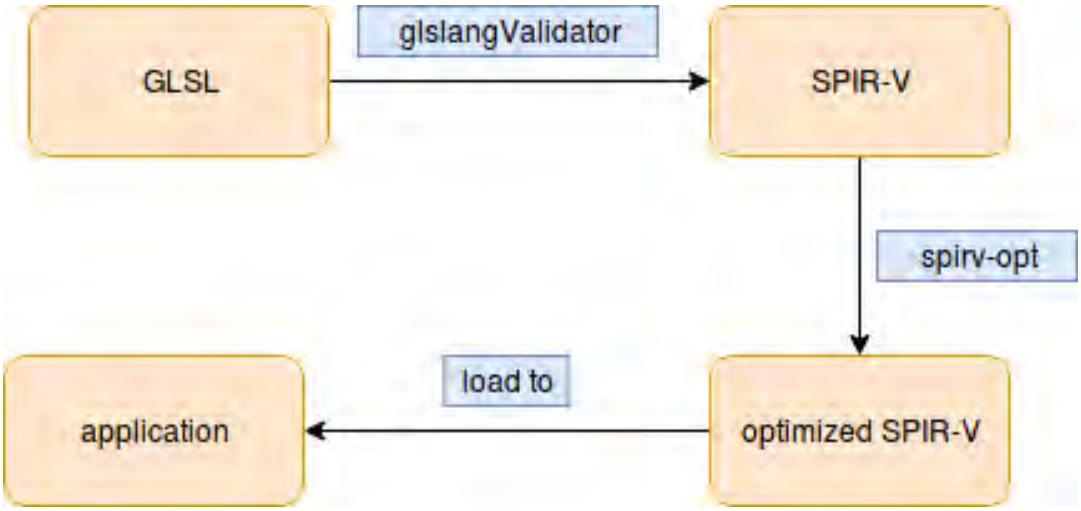


FIGURE 4.3: Optimization process with spirv-opt.

Up to this point, we have described the importance of the compiler to the performance of our implementation. We can measure the computation performance of the implementation in Vulkan and CUDA with all the optimization flags enabled. The performance for both APIs is shown in Figure 4.4.

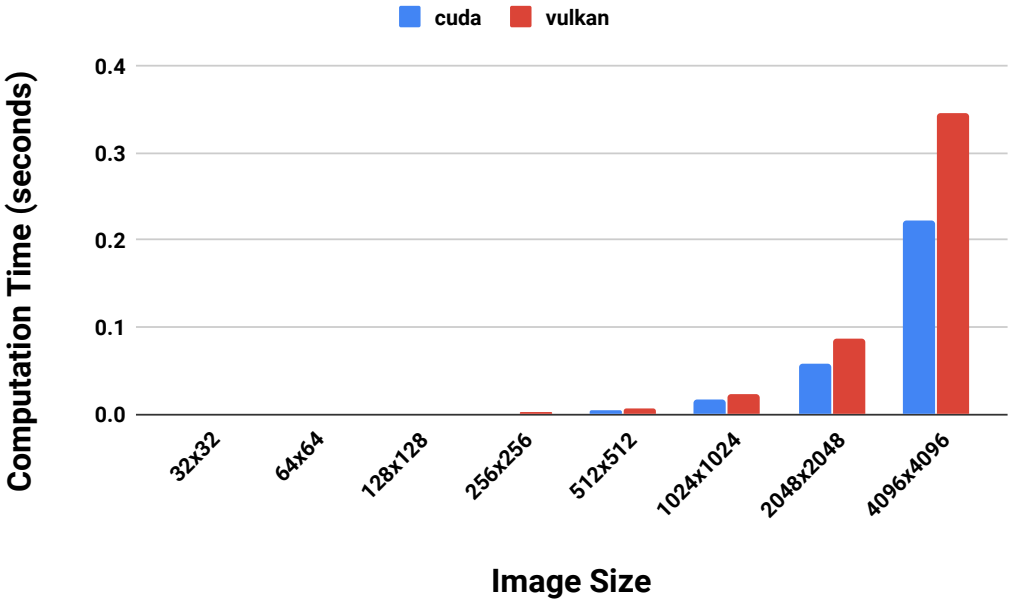


FIGURE 4.4: Computation performance in Vulkan and CUDA with compilers optimization flags on.

As we can see, the computation time for small images is almost zeor for both APIs. On the contrary, the difference in time for big images between Vulkan and CUDA is evident. The implementation of the compute shaders (Vulkan) and kernels (CUDA) is the same, so we expected nearly the same compute performance. However, the compilers have the last word in the performance. In our case, the optimizations of

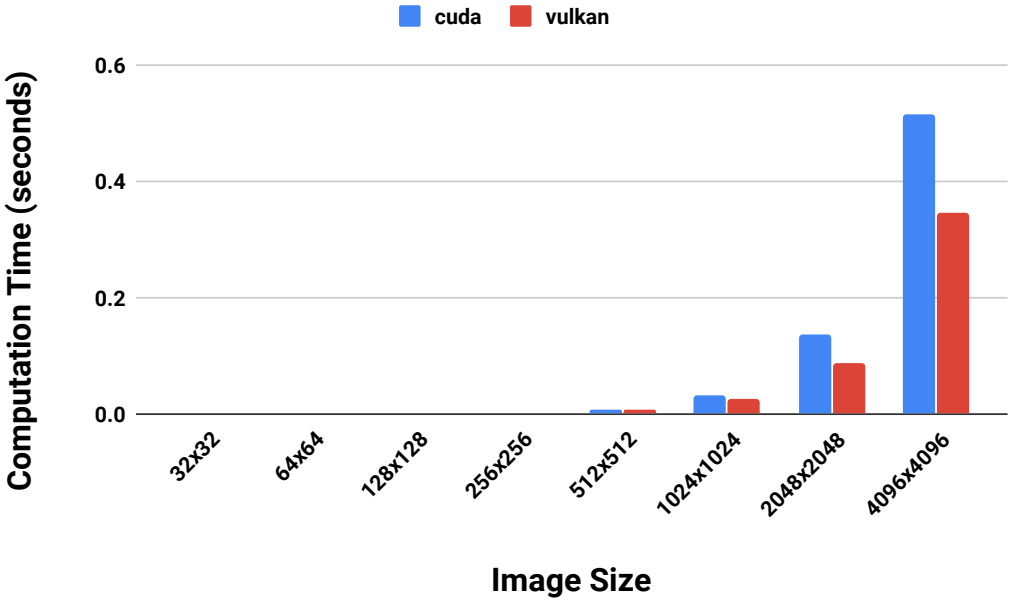


FIGURE 4.5: Computation performance in Vulkan and CUDA with disabled compiler optimizations.

NVCC are very efficient to the kernels, in comparison to spirv-opt, which makes only binary size reduction.

In order to see the baseline performance of both APIs, we disabled the optimizations for NVCC, and we did not use spirv-opt. We can disable all optimizations of NVCC with *-O0 -Xcicc -O0 -Xptxas -O0* arguments during the compilation. Figure 4.5 shows the baseline performance.

Again, for small images, the computation time is zero for both APIs. The critical part here is the computation time on big images. We observe that Vulkan outperforms CUDA when the compiler optimizations are disabled. This reveals the impact of a highly optimized compiler such as NVCC in the performance of the application.

An interesting and unexpected finding is that Image Convlution is faster - when compiler optimizations are disabled - in Vulkan than in CUDA. The absence of profiling tools for compute applications in Vulkan does not allow us to be sure why it is faster than CUDA. We can only make assumptions. Given that Vulkan is a close-to-metal API, it provides a significant degree of freedom to the developer. For instance, as we have already mentioned in our examples before, the developer is responsible for allocating the right type of memory for the application. In CUDA, all these capabilities are hidden within the NVCC compiler.

4.3 NVCC Optimizations

We discussed the quantitative effect of compiler optimizations on the compute performance of the application. In this section, we will explore what kind of optimizations the NVCC applies to the implementation in order to achieve such performance.

Firstly, we should understand what type of application is Image Convolution, namely whether it is memory bound or compute bound. An application is compute bound when the limiting factor of its performance is computation throughput. On the other side, a memory bound application is an application which spends most of its execution time waiting for data to be transferred to it from remote levels of the memory hierarchy.

In our case, Image Convolution can be categorized as a memory bound application. We described the implementation of Image Convolution in Listing 3.3 and 3.4. We can understand from the above code that the computation part of Image Convolution is simply an addition and a multiplication. As a result, each thread which is assigned to calculate the new intensity value of a pixel is waiting for the filter and the neighbor pixels to be loaded. We can understand that the performance of Image Convolution is limited by the number of flops per byte transferred. This is the reason that Image Convolution can be considered as memory bound application.

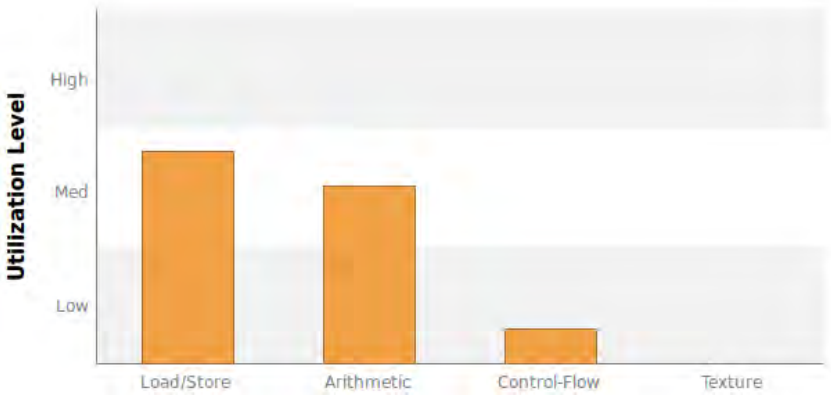
In order to get more details on what kind of optimizations the NVCC can apply, we need to use a profiling tool. We use *NVVP*, the profiler provided by NVidia for CUDA codes.

We compile our implementation twice. Once with the optimizations enabled and once with the optimizations disabled. Since Image Convolution is a memory bound algorithm, we are mostly interested in how the NVCC utilizes the memory. Table 4.2 shows the achieved memory bandwidth with optimizations enabled and disabled.

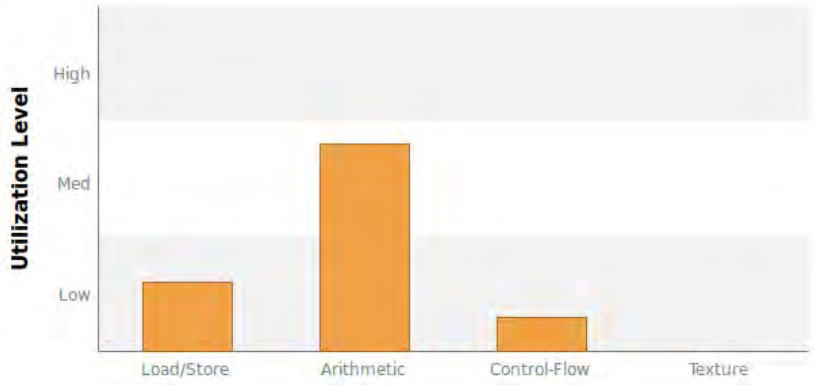
	Optimizations Enabled		Optimizations Disabled	
Memory	Transactions	Bandwidth (GB/s)	Transactions	Bandwidth (GB/s)
L1 Cache	68550656	58.349	68550656	18.406
L2 Cache	172163072	58.349	172163072	18.406
Global	8507167	2.883	9381409	1.003

TABLE 4.2: Memory transactions and bandwidth with enabled and disabled optimizations.

Table 4.2 shows the difference in memory bandwidth between the optimized and un-optimized version of Image Convolution is tremendous. Especially, for L1 and L2 Cache, the memory bandwidth on the optimized version is three times higher that of the non optimized version.



(A) Optimized CUDA Image Convolution (row-wise).

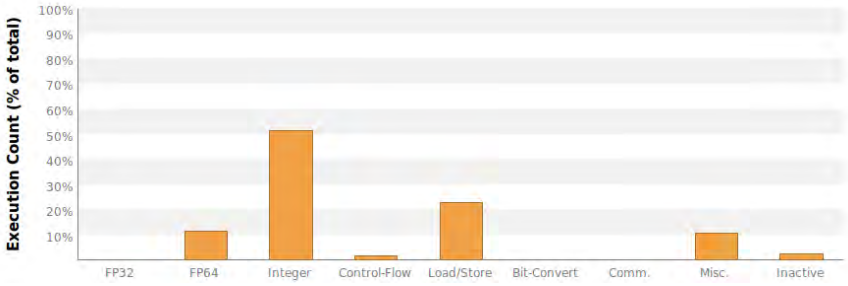


(B) Non optimized CUDA Image Convolution (row-wise).

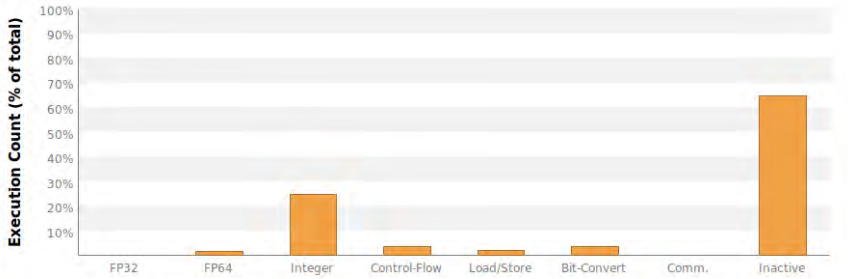
FIGURE 4.6: Function Unit Utilization

How can NVCC achieve such memory bandwidth? Figure 4.7 shows the percentage of thread execution cycles that were devoted to execute all the different types of instructions. We can observe that NVCC utilizes the inactive threads during the execution, in order to load and store memory while other threads are doing computation. This holds true also for Figure 4.6 which depicts the utilization level for different function units. As we can see, NVCC has a significant impact on the utilization of loads and stores operations. In this way, memory bandwidth is increased.

Apart from the memory bandwidth, NVCC can optimize the operations that a kernel is going to execute. Figure 4.8 shows the percentage of thread execution cycles that were devoted to execute different operations. As we can observe, NVCC utilizes the hardware unit called *multiplier–accumulator* which can perform an addition and an multiplication in one step (FUSED MULTIPLY-ADD). This optimization allows the Image Convolution to achieve higher number of flops and as result it increases the computation performance.



(A) Optimized CUDA Image Convolution (row-wise).



(B) Non optimized CUDA Image Convolution (row-wise).

FIGURE 4.7: Instruction Execution Count

(A) Optimized CUDA Image Convolution (row-wise).

(B) Non optimized CUDA Image Convolution (row-wise).

FIGURE 4.8: Floating-Point Operation Counts

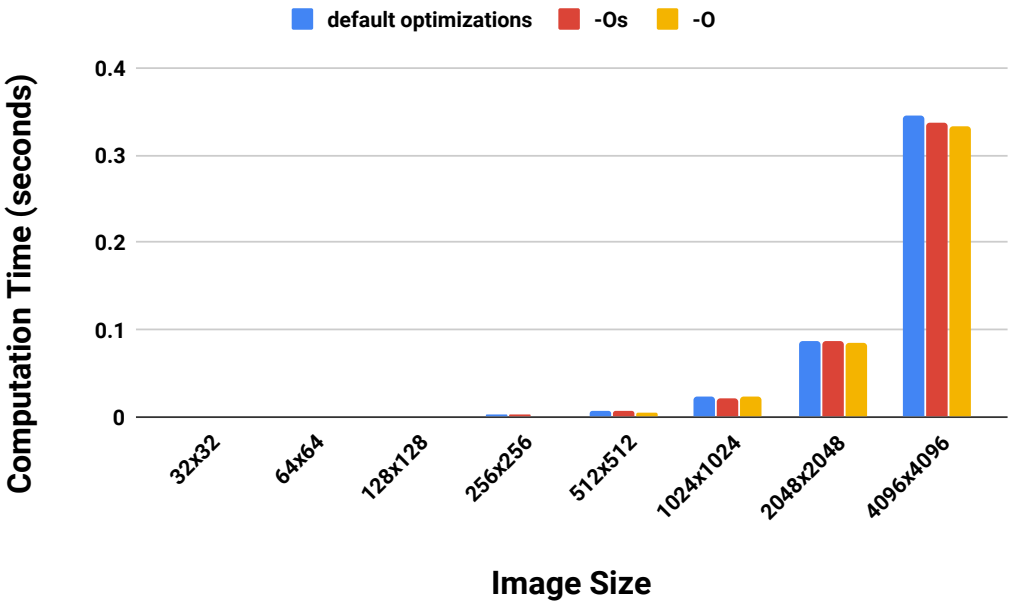


FIGURE 4.9: Image Convolution with default, -O, -Os optimizations from spirv-opt.

4.4 SPRIV-OPT Tool Optimizations

While NVCC plays a vital role in the performance of the Image Convolution, spirv-opt is not up to the task. As we have already mentioned, we use spirv-opt in order to optimize the SPIR-V binary, which is generated from glslangValidator. Spirv-opt can support two optimization flags at the moment. The first one is -O and the second is -Os.

In order to understand the impact of spriv-opt on the performance of the Image Convolution kernel, we ran the algorithm three times. Once with default optimizations, once with -O flag enabled, and finally once with -Os. Figure 4.9 summarizes the impact of spriv-opt (default optimizations, -O, -Os) in the computation time.

As we expected, the spirv-opt is not ready yet to fully optimize compute applications. As we mentioned before, it mainly supports binary file size reduction. As we can observe from Figure 4.9, the size of each of the generated SPIRV binaries is 2756 bytes while the computation time around 0.35 seconds. After the spirv-opt, the size of the binaries is decreased to 2272 bytes while the computation time remains almost the same (0.34seconds).

4.5 Summary

In this chapter, we analyzed the overall and the compute performance of Image Convolution in Vulkan and CUDA. We discussed the development tools we used

for each of the implementations. Additionally, we observed the impact of a highly optimizing compiler such as NVCC to the performance of the application. We also showed the lack of an optimizer for SPIR-V binaries due to the Vulkan toolchain being developed recently. The reason behind the lack of an optimized toolchain for Vulkan is because Vulkan is cross-platform and as a result it is challenging to develop an efficient toolchain for every device.

In the next chapter, we will try to reverse engineer the generated SPIR-V binaries, and we will investigate whether it is possible to optimize them through the NVCC compiler pipeline.

Chapter 5

Compute Shader Binary Optimizations

5.1 Ideas on optimizing a SPIR-V binary

In the previous chapter, we introduced the spirv-opt tool, which is responsible for optimizing SPIR-V binary files. We found that spirv-opt is limited only to the size reduction of the SPIR-V file, and it does not apply any other optimizations to compute shaders for the moment. However, a highly capable optimizer can have a significant impact on the performance of the application as we described through the usage of NVCC. We could get better performance of Image Convolution in Vulkan if we could optimize the SPIR-V binary like NVCC optimizes CUDA applications.

Maybe we can get an idea on how to improve the Image Convolution in Vulkan by observing the compilation process of NVCC. Then, we can apply some of these steps to the SPIR-V binaries from Image Convolution in Vulkan. Figure 5.1 shows the intermediate files which are generated by NVCC during the compilation of a CUDA application.

The intermediate file from NVCC is a *PTX* file. PTX is a stable programming model and instruction set for GPGPU released by Nvidia [19]. In our example, the PTX file contains the intermediate representation(PTX) of the row-wise and the column-wise kernel of Image Convolution in which NVCC applies several optimizations, as we have already mentioned. Since NVCC can apply optimizations in PTX files, one idea is to try converting the SPIR-V binaries directly to PTX in order for NVCC to optimize them. Unfortunately, there is no such tool yet.

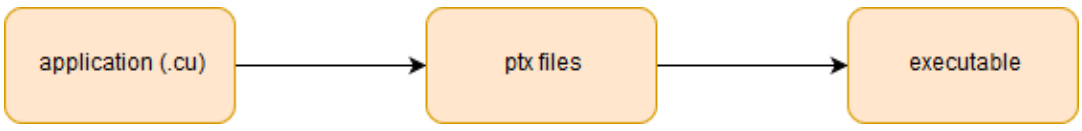


FIGURE 5.1: Generation of intermediate files during the compilation of an application from NVCC.

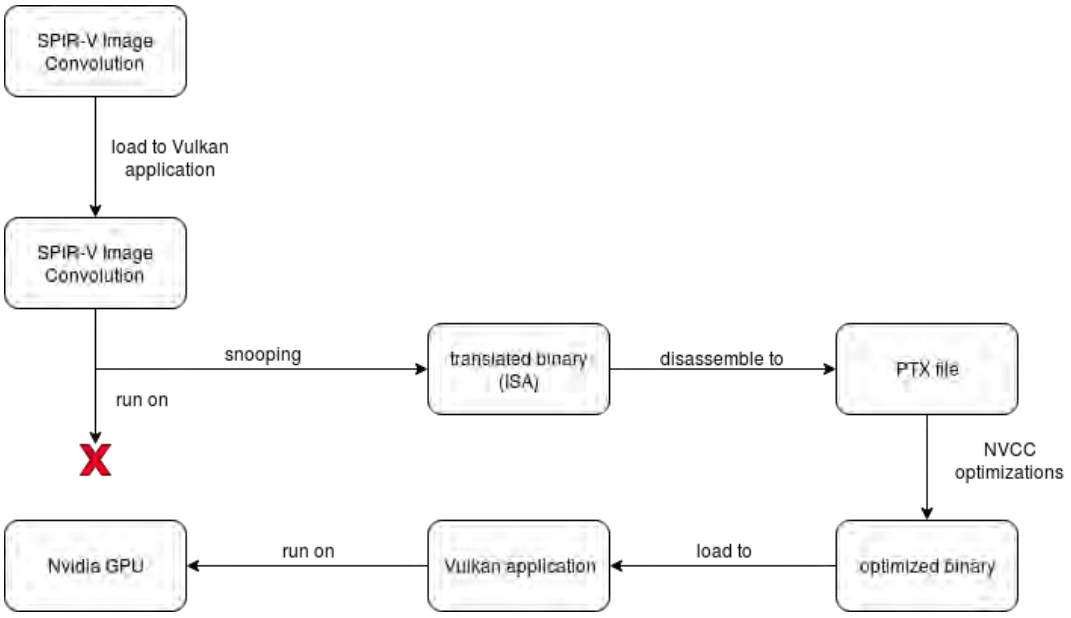


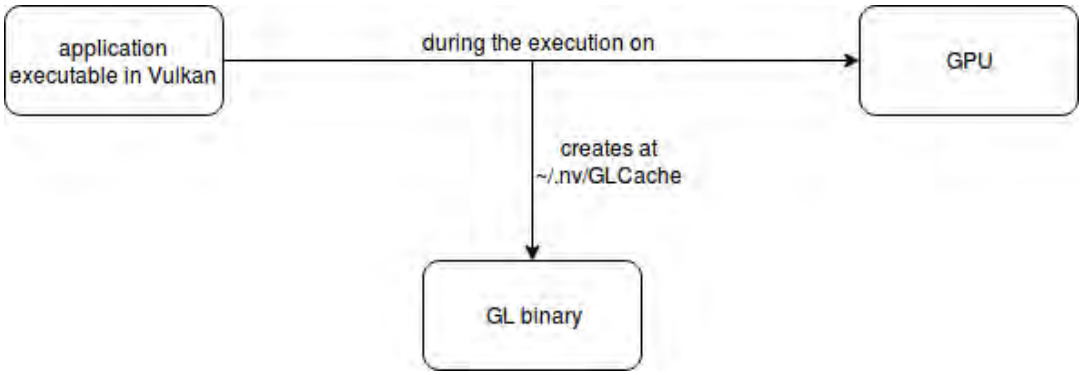
FIGURE 5.2: Optimization of SPIRV through NVCC.

The other file which NVCC generates is executable. The executable file is a binary code which follows the instruction set of the GPU. The GPU cores can execute only instructions which are following the instruction set of the GPU vendor (ISA). The translation from a high-level language to the ISA of the device is the responsibility of compilers (e.g., NVCC for CUDA applications). NVCC applies optimizations to the ptx files during the translation of PTX files to the final machine code (Figure 5.1),

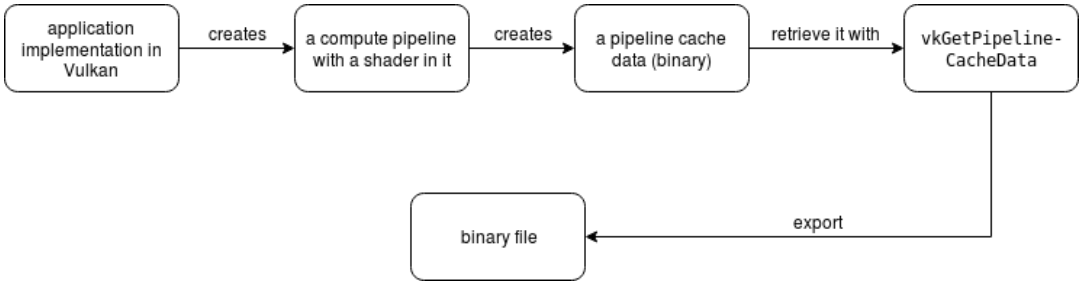
Another idea in order to improve the performance of Image Convolution in Vulkan is to try to disassemble the machine code which is generated by SPIR-V binaries to an earlier form in which NVCC or any other optimizer will be able to apply optimizations. For example, an earlier form of the machine code from SPIR-V binaries can be a PTX file. Then, theoretically, NVCC will most likely be able to optimize the PTX file. Figure 5.2 summarizes the above idea.

5.2 Binary Snooping

In order to check if it is possible to disassemble the machine code from SPIR-V binaries to an earlier form is to try retrieving the machine code of the SPIR-V binaries. As we mentioned, at the end of the day, every executable which is about to run on a device follows the ISA of the device. Image Convolution in Vulkan is no exception. At some point in the compilation of Image Convolution, Vulkan translates the SPRI-V binaries to the appropriate machine code of the GPU. This translation most likely happens in the drivers of Vulkan, and it is hidden from the developer. The developer is responsible only for generating SPIR-V binaries. The question is whether we can



(A) The process of retrieving GL binary.



(B) The process of retrieving pipeline binary.

FIGURE 5.3: GL and pipeline binary.

snoop the translated binary from the application in Vulkan and try to disassemble it?

The Vulkan API allows us to retrieve some sort binaries, which can likely be the binaries from the compute shaders(SPIR-V binaries) that we are interested in. The binary retrieval can be done in two ways, as is shown in Figure 5.3.

The first time we ran the Image Convolution, the executable exported a binary in the home directory under the path *'/.nv/GLCache/'*. The Vulkan API does not export a new binary every time we run the same executable. The above binary is used Vulkan as cached data for the initialization of it. Specifically, this initialization can be the translation from SPIR-V binary to the assembly of the GPU. So, in order to prevent the translation of the same SPIR-V binary to GPU machine code over and over again, Vulkan exports it to a file which is used as a cache. This file is one of the two binaries we can retrieve from the Vulkan API. From now on, we will refer to that binary as GL binary.

Vulkan API generates a binary from the compute pipeline cache. The Vulkan API supports a cache for the creation of pipelines through *VkPipelineCache* and *vkGet-PipelineCacheData*. For example, we can bind a compute shader to a compute pipeline and then create many compute pipelines with the same compute shader with the help of the pipeline cache instead of paying the overhead of pipeline creation all over again. We have access to the pipeline cache through the above API calls. As a result, we can dump the cached binary from the pipeline to a file. Like the GL binary,

we can assume in this case as well that the binary from the pipeline can be related with the compute shader binary since we bind the SPIR-V binary to a pipeline. By comparing the GL binary and the binary from the pipeline, we can observe some similarities. More specifically, the binary from the pipeline can be found entirely inside the GL binary file.

In our Image Convolution implementation, we use two compute pipelines, one for each compute shader. Therefore, we can retrieve two binaries, one for each pipeline. For the sake of simplicity, we will study only one of the two generated binaries. From now on, we will refer to the binary from the compute pipeline as pipeline binary.

5.3 GL and Pipeline Binary

The next step is to investigate if the GL and the pipeline binaries are related to the machine code of SPIR-V binaries. In order to have a better understanding of what these binaries are and how Vulkan API generates them, we need to answer the following questions:

- Is Vulkan or Nvidia driver responsible for the generation of the binaries?
- Is NVCC involved in the generation of the binaries, since it is responsible for generating executables for Nvidia GPUs?

In the current section, we will try to answer these questions.

5.3.1 Nvidia drivers and binaries

In the interest of investigating the association between these binaries and Nvidia drivers, we have to understand what the Nvidia drivers are used for. Nvidia drivers are necessary for running an application on an Nvidia GPU. The drivers act as a gateway between the application and GPU resources.

We mentioned that Image Convolution in Vulkan has to transform the SPIR-V binaries to machine code. This most likely happens inside the Vulkan drivers. However, Khronos Group Inc., which orchestrates and releases the Vulkan, does not release Vulkan drivers for any GPU. GPU vendors provide Vulkan drivers. For example, Nvidia integrates the Vulkan drivers inside the Nvidia GPU drivers for supporting the Vulkan API on their GPUs. We mentioned the specific drivers in Table 4.1. Consequently, the GL or the pipeline binary is likely to be generated from the Vulkan drivers; hence, the Nvidia drivers. Unfortunately, Nvidia drivers are closed-source (proprietary), and therefore, we cannot validate if the drivers generate the binaries or not.

To further investigate GPU binary generation from Vulkan codes, we resort to OS monitoring tools. Applications can interact with the Linux OS through *system calls*. The Linux OS supports snooping systems calls performed by an application with the *strace* command. If the Image Convolution implementation in Vulkan communicates with the Nvidia drivers at some point of its execution, then this interaction has to happen through the OS.

We confirm the above allegation from the output of strace for the Image Convolution. In more detail, the output from strace shows that the Image Convolution calls *ioctl* commands in order to interact with the Nvidia drivers. ioctl is a system call which is responsible for manipulating underlying devices parameters through associated pseudo-files. Consequently, we can assume that Nvidia drivers generate the GL binary. Since the GL binary contains the pipeline binary, it is wise to assume that the Nvidia drivers create the pipeline binary in the same way.

5.3.2 NVCC and binaries

One assumption could be that Nvidia drivers generate the binaries by upcalling NVCC. In order to test this assumption, we tried to run the Vulkan version of Image Convolution with the NVCC disabled. We disabled NVCC by removing its permissions for reading, writing, and executing from the system. However, the application could run normally. Subsequently, the assumption that NVCC is involved in the creation of the binaries is not valid.

Until now, we have not tried to analyze what data each of these binaries contains. That said, we converted the binaries to ASCII characters. Figure 5.4 shows the results from the conversion. As we can see in the red box from the figure, there is a sequence of four characters formatting the word *NVVM*.

NVVM is a compiler IR [18]. Compilers use an Intermediate Representation (IR) language in order to represent the source code internally. The compilers apply optimizations to the IR. Specifically, NVVM IR is designed for representing GPU compute kernels such as CUDA kernels. Additionally, NVVM IR is a subset of the LLVM IR, which is another common intermediate representation used by the LLVM compiler framework.

The NVVM label in Figure 5.4 could be an indicator that the pipeline or GL binary file is the compute shader in the NVVM IR binary format. We could get more details of GL or pipeline binary, if we could compare them with the NVVM IR format of row-wise or column-wise Image Convolution CUDA kernels. Unfortunately, due to Nvidia closed-source development tools, NVCC cannot expose the NVVM IR binary format from CUDA kernels.

On the bright side, NVCC is not the only compiler which can compile CUDA kernels. There is another option for compiling CUDA kernels, albeit not as well optimized as NVCC, which is the *clang*++ compiler. clang++ is capable of exposing LLVM IR. As we mentioned, NVVM IR is a subset of LLVM IR; hence, by retrieving the LLVM IR of a CUDA application by clang++ can result in retrieving the NVVM IR of the application at the same time. The extraction of NVVM IR using clang++ allows us to use some development tools from LLVM Compiler Infrastructure [22]. We will discuss the usage of LLVM and Nvidia development tools on the GL and pipeline binaries later on.

```
00000000: 2000 0000 0100 0000 de10 0000 9212 0000  ................
00000010: 79bb 6a2d b89f 1d76 b46e 78d8 47f9 7c8b  y.j-...v.nx.G.|.
00000020: 0100 0000 0d26 8bb7 5a85 174e 82b6 a665  .....&..Z..N...e
00000030: 9792 ace9 5502 0000 0000 0000 4350 4b56  ....U.......CPKV
00000040: 0000 0000 0d26 8bb7 5a85 174e 82b6 a665  .....&..Z..N...e
00000050: 9792 ace9 f86f 48b8 3c6c e012 5502 0000  .....oH.<l..U...
00000060: 0000 0000 7c05 0000 094e 5656 4d4e 5675  ....|....NVVMNVu
00000070: 6301 c30b 0600 0800 0800 7000 0500 02c5  c.........p.....
00000080: 02e0 02c6 0103 c301 08c3 01e0 d701 01c3  ................
00000090: 05c0 0100 00f0 d701 45cf 0101 cf01 17cf  ........E.......
000000a0: 0120 c303 0200 01c9 0139 c301 20c3 02b0  . .......9.. ...
000000b0: 02c6 0101 c301 05c3 0101 c701 21c3 0107  ............!...
000000c0: c302 d002 c601 01cf 010c c301 40cb 3f10  ............@.?.
000000d0: a010 809c 1080 082e fc1f c700 0088 6002  ..............`.
000000e0: 009c 1200 0040 8606 001c 1300 0040 8602  .....@.......@..
000000f0: 001c 0400 0080 601e 2c1c c700 1c48 5b0e  ......`.,....H[.
00000100: 009c 1000 0040 8606 049c 0400 0080 3560  .....@........5`
00000110: 9c10 8c10 80ac 8008 0a00 1c11 0000 4086  ..............@.
00000120: 2100 1c10 000c 08a1 0200 9c7f 003c c0e4  !............<..
00000130: 2504 1c10 0008 08a1 0600 9c7f 003c c0e4  %............<..
00000140: 3c00 0078 c311 120a 209c 0500 0080 e09c  <..x.... .......
00000150: b0b0 aca0 10a0 08c3 1860 0000 8014 1d08  .........`......
00000160: 9cff ff1f 48bb 1e08 1cc6 0000 185b 023c  ....H........[.<
00000170: 60c3 3f80 850a 241c c600 0808 5119 089c  `.?...$.....Q...
00000180: 0100 0040 c20a 2c1c c700 0090 6080 a088  ...@..,.....`...
00000190: b0a0 10b0 081e 181c 0700 1c60 5b09 089c  ...........`[...
000001a0: 0100 0040 c23e 081c 0b00 1c60 5b1a 1820  ...@.>.....`[..
000001b0: 0600 c115 8460 1efc a306 0040 8060 1208  .....`.....@.`..
000001c0: 240a 0000 8460 0818 20c3 1580 c5a0 fc10  $....`.. .......
000001d0: 80ac 10ac 0816 fca7 0a00 4080 6010 1024  ..........@.`..$
000001e0: c316 80c5 0a00 807f 003c c0e4 0e00 807f  .........<......
000001f0: 003c c0e4 3c00 2408 c342 12c1 219c 7f00  .<..<.$..B..!...
00000200: 3cc0 e416 009c 7f00 3cc0 e4ac a0b0 b8a0  <.......<.......
00000210: a0b0 0802 105c 0100 0080 db2d 2c9c c432  .....\.....-,..2
00000220: 401e 2c1c c700 1c48 5b3c 0020 88ff 7f00  @.,....H[<. ....
00000230: 1222 241c c600 2008 5121 209c 0100 0040  ."$... .Q! ....@
00000240: c21e 201c 0900 1c60 5bb8 b0a0 10b8 0000  .. ....`[.......
00000250: 083c c614 180a 201c 0800 0084 600e fc9f  .<.... .....`...
00000260: 0800 4080 6000 081c c305 80e5 3c00 1cc4  ..@.`.......<...
00000270: 0c18 3c00 1cfc ff7f 0012 023c 1cc3 0280  ..<........<....
00000280: 85c8 0601 0001 00c0 01c9 0180 c806 6d61  ..............ma
00000290: 696e 5f34 ffc1 0301 0020 c301 20c3 0101  in_4..... .. ...
000002a0: d701 01c3 0102 c301 64d3 bfbf 82cc bfbf  ........d.......
000002b0: 82cc bfbf 82cc bfbf 82                   .........
```

FIGURE 5.4: ASCII representation of pipeline binary.

5.4 Binaries Disassembly

We showed how to retrieve the GL and pipeline binary from the application. We also showed the correlation of these binaries with the Nvidia GPU and development tools. In order to test whether GP or the pipeline binaries include machine code for the GPU, we tried to disassemble them. Nvidia provides CUDA binary utilities [2] such as *cuobjdump* and *nvdisasm*, which are responsible for disassembling CUDA applications.

Unfortunately, we tried both tools on the GL and pipeline binary without any success. More specifically, both tools could not recognize the binaries as a device code, so they could not proceed with the disassembling. We also tried to remove several numbers of bytes from the beginning of both binaries in order to check if there are extra headers in them which prevent the tools from disassembling. However, the disassemblers could not proceed with the disassembling. We are not able to get more details on how these binaries are formed because as we mentioned, Nvidia drivers are proprietary.

Since the disassembling of the binaries failed with the tools of Nvidia, we can try to disassemble them thought LLVM tools. The LLVM Compiler Infrastructure provides the *llvm-dis* disassembler. As with Nvidia tools, llvm-dis could not disassemble the two binaries as well. It returned *Invalid bitcode signature* for both binaries.

5.5 Summary

In this chapter, we investigated whether GP or the pipeline binaries include machine code for the GPU. First, we showed that the SPIR-V binaries have to be translated to machine code, which is following the GPU ISA of the system, in order to be able to run on the GPU. Based on that, we confirmed that the Vulkan application is related to the Nvidia drivers.

Besides, we showed how to retrieve the GL and the pipeline binary. We tried to analyze them and get information on how they are generated or what is included in them. Finally, hypothesizing that the two binaries may contain GPU machine code, we tried to disassemble these binaries with the usage of cuobjdump, nvdisasm, and llvm-dis, but unfortunately, they all failed. Due to Nvidia drivers being closed-source, we were not able to get more details about GL and pipeline binary. However, through this disassembling failure, we obtained a better picture of what is a SPIR-V binary, how Vulkan handles it, what code runs on the GPU and why code optimizers are so crucial in the performance of an application.

Chapter 6

LLVM IR and clspv Optimizations

The idea here is the same as in Chapter 5. We will try to convert the SPIR-V binaries to other forms on which we will be able to apply optimizations. Later, if the above try succeeds, we will try convert the optimized code back to SPIR-V.

6.1 LLVM IR

Our first try in order to increase the performance of Image Convolution is by testing if we transform the SPIR-V binaries into LLVM IR and then optimize them through llvm-opt. There is a tool capable of converting SPIR-V binaries to LLVM IR and vice versa. The tool is called *LLVM/SPIR-V Bi-Directional Translator* and it can be found on Github.

Although we experimented with the LLVM/SPIR-V Translator on the SPIR-V compute shaders, we could not get any results. More specifically, the LLVM/SPIR-V Translator could not translate the SPIR-V binaries. The reason why it failed on translating the SPIR-V binaries is that it does not support SPIR-V binaries which are generated from GLSL code. Instead, it supports SPIR-V binaries, which are generated from OpenCL. We tried to solve this problem by changing the headers from the SPIR-V binaries in order for the binaries to look like they are generated from OpenCL kernels, but still, the LLVM/SPIR-V Translator could not translate the binaries successfully.

6.2 clspv

This inability of LLVM/SPIR-V Translator motivated us to try to optimize the SPIR-V compute shaders for Image Convolution in another way. More specifically, we will implement the Image Convolution compute shaders in OpenCL and then compile them to SPIR-V binary. Listing 6.1 and 6.2 show the row-wise and column-wise of Image Convolution respectively in OpenCL.

```
__attribute__((reqd_work_group_size(32,32,1)))
#define N 4096
__kernel void rowShader(__global double *d_Input, __global double *
    d_Output, __global double *d_Filter) {
    unsigned int x = get_global_id(0);
    unsigned int y = get_global_id(1);

    int k, d;
    double sum = 0.0;

    for (k = -16; k <= 16; k++) {
        d = (int)x + k;
        if (d >= 0 && d < N) {
            sum += d_Input[(int)y * N + d] * d_Filter[16 - k];
        }
    }
    d_Output[y * N + x] = sum;
}
```

LISTING 6.1: Convolution over rows OpenCL

```
__attribute__((reqd_work_group_size(32,32,1)))
#define N 4096
__kernel void colShader(__global double *d_Input, __global double *
    d_Output, __global double *d_Filter) {
    unsigned int x = get_global_id(0);
    unsigned int y = get_global_id(1);

    int d, k;
    double sum = 0.0;

    for (k=-16; k<=16; k++) {
        d = (int) y + k;
        if (d>=0 && d < N) {
            sum += d_Input[d*N+(int)x] * d_Filter[16-k];
        }
    }
    d_Output[y * N + x] = sum;
}
```

LISTING 6.2: Convolution over columns OpenCL

There is no difference inside the core of the Image Convolution implementation (for loop) between the Listing 6.1 (OpenCL), 3.1 (Vulkan) and 3.3 (CUDA) for the row-wise kernel and between the Listing 6.2 (OpenCL), 3.2 (Vulkan) and 3.4 (CUDA) for the column-wise kernel. Apart from the for loop, all the other code is different in OpenCL, GLSL, and CUDA because each language uses different semantics and definitions. This divergence outside of the for loop between the Listings above reveals how each language defines a kernel or compute shader. Although this does not affect the total performance of Image Convolution.

We are ready now to convert the OpenCL implementation to SPIR-V binary and load it to the Vulkan application. To do this, we used *clspv*. This tool is a prototype compiler for a subset of OpenCL C to Vulkan compute shaders and can be found on Github. In other words, clspv can translate the OpenCL Image Convolution kernels to Vulkan compute shaders. Moreover, it applies a set of optimizations during translation. We can observe the performance of Image Convolution using glslangValidator, clspv and NVCC in Figure 6.1(optimization enabled) and 6.2(optimizations disabled).

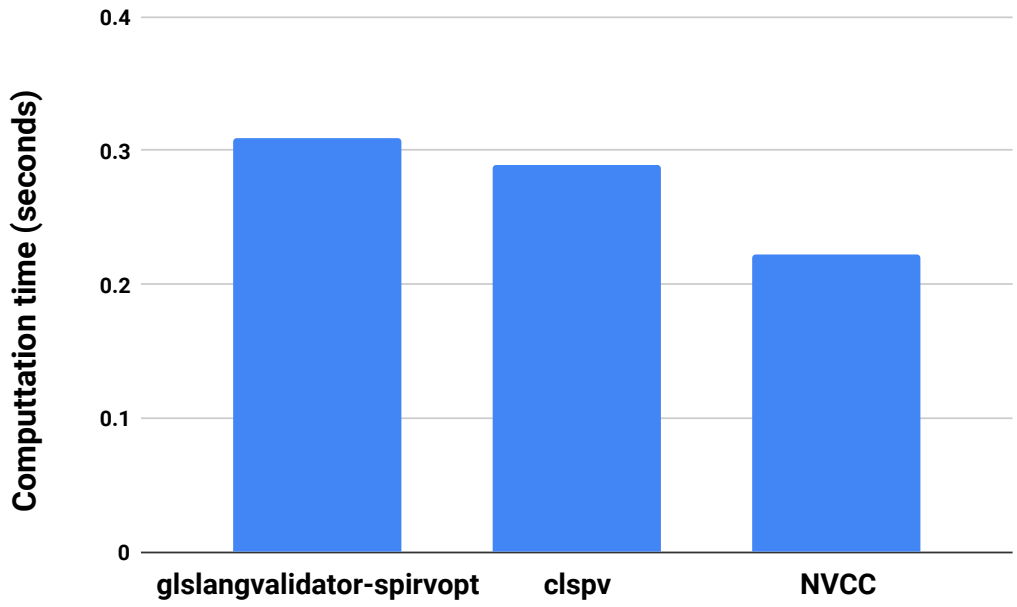


FIGURE 6.1: Compute Performance of Image Convolution (4096x4096 Image size) with optimizations enabled for the combination of glslangValidator and spirv-opt, clspv and NVCC.

We can observe a performance gain between the glslangValidator-spirv-opt and the clspv version of Image Convolution. The performance gain of the clspv version of Image Convolution is because clspv applies a set of optimizations to the OpenCL code; hence, to SPIR-V binaries. However, these optimizations cannot be compared with the optimizations which are applied from NVCC to the CUDA Image Convolution. We can confirm this from Figure 6.1 where NVCC outperforms both glslangValidotor-spirv-opt and clspv. We have already explained why NVCC is so powerful in Chapter 4.

On the contrary, when the optimizations are disabled for all of the above compilers, glslangValidator achieves a slightly better performance in comparison to clspv and a significant performance gain in comparison to NVCC as we have already mentioned. The difference in the performance of Image Convolution with glslangValidator and clspv is negligibly small, and it can be ignored. We can assume that glslangValidator is better implemented than clspv due to the previous results.

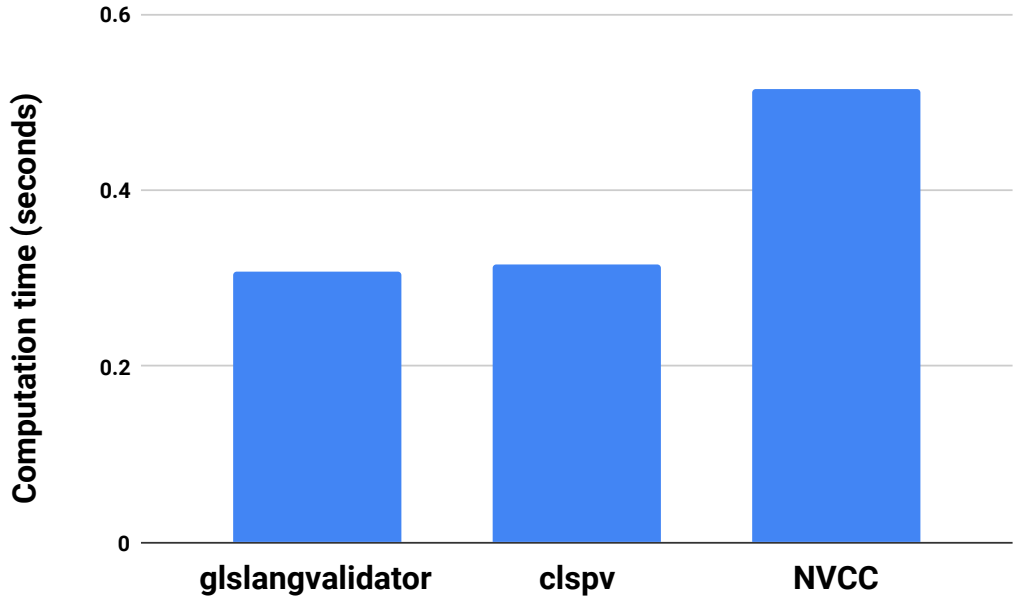


FIGURE 6.2: Compute Performance of Image Convolution (4096x4096 Image size) with optimizations disabled for glslangValidator, clspv and NVCC.

6.3 Summary

In this chapter, we continued the investigation on how to increase the performance of Image Convolution in Vulkan. First, we tried to translate the SPIR-V binaries to LLVM IR, but LLVM/SPIR-V Translator is not supporting SPIR-V binaries, which are generated from GLSL code. This failure led us to another way of optimizing the SPIR-V binaries. We implemented the Image Convolution in OpenCL and used clspv in order to generate optimized SPIR-V compute shader binaries. We observed that clspv achieves a better performance in comparison to the combination of glslangValidtor and spirv-opt.

Chapter 7

Conclusion

In this thesis, firstly, we introduced the Vulkan API by implementing the vector addition algorithm. This example allowed us to understand the necessary steps we need to follow for implementing any compute application in Vulkan. Next, we described the Image Convolution algorithm, which we used for evaluating the performance of Vulkan and CUDA. We showed the highly parallelizable nature of Image Convolution, and we implemented in Vulkan and CUDA.

We observed that Image Convolution could perform better in CUDA than in Vulkan when the compiler optimizations are enabled. On the other hand, Vulkan outperformed CUDA when we disabled the compiler optimizations. We observed also that NVCC is a highly optimized compiler for CUDA applications, and it has a significant impact on the performance of Image Convolution when the optimizations are enabled. While glslangValidator and spirv-opt, due to their recent release, they could not improve the performance of Image Convolution in Vulkan.

The need for improving the performance of Image Convolution in Vulkan led us on investigating the generated machine code of SPIR-V binaries with the purpose of optimizing through NVCC. We tested if the generated binaries from the Vulkan application contain machine code, but the results were negative. We also tried to convert the SPIR-V binaries to LLVM IR and then optimize the LLVM IR code, but without any success either. Finally, we implemented the Image Convolution in OpenCL, and then we converted the Image Convolution kernels to SPIR-V binaries through clspv. We achieved slightly better performance from clspv than the glslangValidator and spirv-opt. However, the SPIR-V binaries from clspv could not compete with the performance achieved by CUDA and NVCC.

In conclusion, due to Vulkan and its development tools being recent, it cannot compete with the performance of CUDA and NVCC on Nvidia GPUs. However, it would be interesting if an optimized compiler for GLSL or SPIR-V binary was released soon in order to increase the performance of compute applications in Vulkan.

In order to further investigate the performance of Vulkan compute applications, a profiler for SPIR-V binaries is required. Right now, we are not able to get a clear view

of how the they perform on the GPU through Vulkan, but a performance profiler will be very useful in the future.

In the current thesis, we evaluated the performance of Vulkan on Nvidia GPUs. However, they not the only GPU vendor on the market and testing the same applications on AMD GPUs could provide interesting results. AMD is known about its open-source GPU drivers, and it can allow us to get insight on how the machine code is generated from SPIR-V binaries. As a result, we can have more information on how the SPIR-V binaries perform, and it can lead in new ways of optimizing them.

Bibliography

[1] *CUBLAS LIBRARY*. DU-06702-001 v10.1. User Guide. NVIDIA. May 2019.

[2] *CUDA BINARY UTILITIES*. DA-06762-001 v10.1. Application Note. NVIDIA. May 2019.

[3] *CUDA COMPILER DRIVER NVCC*. TRM-06721-001 v10.1. Reference Guide. NVIDIA. May 2019.

[4] *CUDA-MEMCHECK*. DU-05355-001 v10.1. NVIDIA. May 2019.

[5] *CUDA RUNTIME API*. vRelease Version. NVIDIA. June 2019.

[6] *CUSOLVER LIBRARY*. DU-06709-001 v10.1. NVIDIA. May 2019.

[7] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. "A comprehensive performance comparison of CUDA and OpenCL". In: *2011 International Conference on Parallel Processing*. IEEE. 2011, pp. 216–225. DOI: 10.1109/ICPP.2011.45.

[8] Maria Rafaela Gkeka. "Porting the Laplacian filtering application to the Vulkan API using OpenCL and OpenGL programming models". MA thesis. University of Thessaly, Oct. 2018.

[9] Fischer Greg. *SPIR-V Legalization and Size Reduction with spirv-opt*. LunarG Inc. May 2018.

[10] The Khronos Vulkan Working Group. *Vulkan 1.1.110 - A Specification*. v1.1.110. Khronos Group. June 2019.

[11] Apple Inc. *Performing Convolution Operations*. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html.

[12] John Kessenich, Boaz Ouriel, and Raun Krisch. *SPIR-V Specification*. v. 1.4. Revision 1. Khronos Group. May 2019.

[13] LunarG. *SPIR-V Toolchain*. https://vulkan.lunarg.com/doc/sdk/1.1.106.0/linux/spirv_toolchain.html.

[14] Segal Mark and Akeley Kurt. *The OpenGL Graphics System:A Specification*. Version 4.6 (Core Profile). The Khronos Group Inc. Feb. 2019.

[15] Aaftab Munshi. "The opencl specification". In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE. 2009, pp. 1–314. DOI: 10.1109/HOTCHIPS.2009.7478342.

[16] *NSIGHT COMPUTE*. v2019.3.1. User Manual. NVIDIA. May 2019.

[17] *NVBLAS LIBRARY*. DU-06702-001 v10.1. User Guide. NVIDIA. May 2019.

[18] *NVVM IR SPECIFICATION 1.4*. SP-06714-001 v1.4. Reference Guide. NVIDIA. May 2019.

[19] *PARALLEL THREAD EXECUTION ISA*. Application Guide v6.4. NVIDIA. May 2019.

[20] *PROFILER USER'S GUIDE*. DU-05982-001 v10.1. NVIDIA. May 2019.

[21] Graham Sellers and John Kessenich. *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016. ISBN: 978-0134464541.

[22] *The LLVM Compiler Infrastructure*. https://llvm.org/.

[23] Akenine-Möller Tomas, Haines Eric, and Hoffman Naty. *Real-Time Rendering*. Fourth Edition. 2018. ISBN: 978-1138627000.

[24] Niblack Wayne. *An Introduction to Digital Image Processing*. Prentice Hal, 1986. ISBN: 978-0134806747.

[25] Wikipedia. *Kernel (image processing)*. https://en.wikipedia.org/wiki/Kernel_(image_processing).